# NAVAL POSTGRADUATE SCHOOL
## Monterey, California




# THESIS

AN ANALYSIS OF ALIASING IN
BUILT-IN SELF TEST PROCEDURE

by

Jasa Barus

June 1991

Thesis Advisor: Chyan Yang

**Approved for public release
distribution unlimited.**

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | |
|---|---|
| 1a REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b RESTRICTIVE MARKINGS |
| 2a SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION / AVAILABILITY OF REPORT<br>public release; distribution is unlimited |
| 2b DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S) | 5 MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a NAME OF PERFORMING ORGANIZATION<br>Naval Postgraduate School<br>6b OFFICE SYMBOL (If applicable) | 7a NAME OF MONITORING ORGANIZATION<br>Naval Postgradute School |
| 6c ADDRESS (City, State, and ZIP Code)<br>Monterey, California 93943-5000 | 7b ADDRESS (City, State, and ZIP Code)<br>Monterey, California 93943-5000 |
| 8a NAME OF FUNDING / SPONSORING ORGANIZATION<br>8b OFFICE SYMBOL (If applicable) | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c ADDRESS (City, State, and ZIP Code) | 10 SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO |

11 TITLE (Include Security Classification)
An Analysis of Aliasing in Built-in Self Test Procedure

12 PERSONAL AUTHOR(S)
Jasa Barus

| 13a TYPE OF REPORT | 13b TIME COVERED | 14 DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| Master's Thesis | FROM _____ TO _____ | 1991 June | 80 |

16 SUPPLEMENTARY NOTATION

| 17 COSATI CODES | | | 18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Built-in Test; Aliasing Probabibility; Test Architecture |
| | | | |
| | | | |

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis investigates aliasing probability in Built-in Self Test (BIST) procedure, in which a Linear Feedback Shift Register (LFSR) is used as pseudo-random pattern generator, with a full adder as Circuit Under Test (CUT). The Signature Analyzer implements a Multiple Input Signature Register (MISR) as a test response compressor.

| | |
|---|---|
| 20 DISTRIBUTION AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | 21 ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED |
| 22a NAME OF RESPONSIBLE INDIVIDUAL<br>Prof. Chyan Yang | 22b TELEPHONE (Include Area Code)<br>(408) 646-2266<br>22c OFFICE SYMBOL<br>EC/Ya |

DD Form 1473, JUN 86 *Previous editions are obsolete*

S/N 0102-LF-014-6603

Approved for public release; distribution is unlimited

An Analysis of Aliasing in Built-in Self Test Procedure

by

Jasa Barus
Captain, Indonesian Air Force
B.S., Padjadjaran University, Indonesia, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN
ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

June 1991

Author:

Jasa Barus

Approved by:

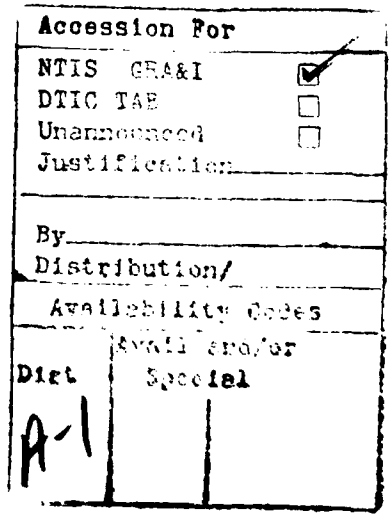
Chyan Yang, Thesis Advisor

Herschel H. Loomis, Jr., Second Reader

Michael A. Morgan
Department of Electrical Engineering

# ABSTRACT

This thesis investigates aliasing probability in Built-in Self Test (BIST) procedures, in which a Linear Feedback Shift Register (LFSR) is used as a pseudo-random pattern generator, with a full-adder as a circuit-under-test (CUT). The Signature Analyzer implements a Multiple Input Signature Register (MISR) as a test response compressor.



# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Professor Chyan Yang and my second reader, Herschel H. Loomis, Jr., for their guidance and assistance. And, also I want to thank Col. Nav Mardjuki, my wife Jenni, and my sons Dea and Andri for their support.

# I. INTRODUCTION

Advances in semiconductor manufacturing technology permit increased complexity of VLSI circuits, and therefore increase the difficulty of testing devices, in particular, those aspects of chip testing such as cost, time, and test data volume.

One approach to alleviate testing problems to reduce both cost of test generation and test time is to incorporate built-in self test (BIST) circuitry into the circuit-under-test during the initial design. A BIST circuit requires that the test pattern generation and output response evaluation are incorporated into the circuit under test. In this technique, test patterns are applied internally to the circuit under test (CUT) and their output responses are evaluated without the use of external test equipment. There are three methods to generate test patterns [Ref. 1]:

- Manual Generation - The test patterns are written by a design or test engineer with regard to the fault coverage.
- Pseudo-random Pattern Generation - Pseudo-random test patterns are generated by hardware built into the test equipment or actually embedded in the CUT or by a software algorithm.
- Algorithmic Test Generation - A computer program is used to generate the test patterns, and also deals with fault coverage.

The method used in BIST for evaluating response is most frequently performed by signature analysis. [Ref. 2]

The most popular approach used for BIST today is pseudo-random pattern generation. Pseudo-random patterns are usually generated by a linear feedback shift

register (LFSR) and are applied to the circuit under test. The output responses from the circuit under test are then compressed through multiple-input shift register (MISR) to form a signature. By analyzing the signature, it can be determined whether a circuit under test is faulty or not. Because test patterns are generated by a built-in random pattern generator, the time consuming effort of test pattern generation can be removed from the design cycle by using a BIST circuit. Since signature analysis compresses response data and compares the signature only once, the difficulty of analysis and storage of huge amounts of test response data can be avoided. Furthermore, BIST has another important advantage in that the circuit under test is fed with random test patterns at the functional clock rate. This advantage allows high speed testing to be performed using internal test equipment built in the circuit under test.

In designing a built-in self test circuit, the following factors need to be considered [Ref. 3]:

- Performance Degradation - The maximum operating frequency may be reduced because of the additional propagation delay caused by the self-test circuit.

- Overhead - The added self-testing circuitry will increase the chip area, and this will increase the manufacturing cost. Therefore, the overhead should be kept as small as possible.

- Fault Coverage - The ratio between the number of faults detected by using the set of test-pattern given is also an indicator of the quality of the test. A coverage between 90% to 95% is desired for satisfactory performance. Basically, fault coverage depends on the effectiveness of the input sequences.

- Testing of the Added Circuitry - The circuit which added to the chip also should be tested.

- Test Application Time - This will affect the final cost.

The test pattern is selected based on the above factors. If a complicated pattern or exhaustive pattern set is selected, the fault coverage could be increased, but circuitry will then become more complex and require more difficult testing parameters. The overhead will increase and more test time may be necessary. If a simple test pattern or subset of exhaustive pattern set is selected, the added circuitry can be easily tested. This results in lower overhead and shorter test time, but lower fault coverage. The major problem of this BIST technique is the possibility that correct and faulty circuits give the same signature, known as aliasing errors. These errors depend on the structure of compressor, the structure of CUT, the type of fault and the test patterns [Ref. 4].

The main purpose of this study is to set guidelines and to evaluate testing strategies used for BIST. In Chapter II, we investigate the testing concepts used in BIST and annotate their advantages and disadvantages. In Chapter III, aliasing probability is discussed. Chapter IV presents testing requirements and tools used in this thesis. Conclusions are given in the last chapter.

# II. BUILT-IN TEST BACKGROUND

## A. TEST GENERATION AND VERIFICATION

Test generation is used to generate a sequence of input test vectors to verify the correctness of the circuit. Test verification measures the effectiveness of a given set of test vectors. A complete set of test vectors may guarantee a 100% fault coverage but may not be practical to execute when the size of test vectors is high. All of these considerations should be taken into account when designing a test strategy. As mentioned in Abadir and Regubati [Ref. 5], test generation should consist of three main activities:

- A good descriptive model should be selected for a system under consideration to determine the behavior of the system in all possible combination modes of operation.

- A fault model must be developed in order to describe the types of faults identified during test generation. The important factor in selecting a fault model is to maximize the percentage of faults covered by the model and to minimize the test cost associated with the use of the model. Usually, a good fault model is found as a result of a trade-off between these two parameters.

- The most essential part of the BIST process is to generate a test sequence to detect all the faults in the fault model. Two major problems occur when generating a test sequence. To detect a certain fault in a digital circuit, the fault must be excited, e.g., a certain test sequence must be applied that will force a faulty value to appear at the fault location. The test must be sensitive to the

fault, e.g., the effect of the fault must be propagated through the network to an observable output.

The fault must be able to be located following its detection. The strategy of testing can be changed depending on whether the requirement to detect the fault only, or to detect and locate the fault. The manual generation of test patterns is a difficult, time consuming task even for moderate circuits. The challenge of fault simulation and test verification has received a lot of attention, while the task of test generation has been partially overlooked. Fault simulation has been the goal of test generation, yielding a quantitative measure of test effectiveness. In other words, a test sequence is considered good if it can detect a high percentage of the possible faults in the circuit under test. Fault simulation should be able to predict how a circuit will operate, therefore, the simulation program must be comprehensive enough to detect it. The fault model and test pattern generation are reviewed below:

- Fault Simulation and Fault Model

  Two major sources of failures occurring in integrated circuits are defects in the manufacturing process and component wearout. The frequency of occurrences and the relative importance of a fault depends on the circuit type and the manufacturing technology used. By using any given physical fault mechanism in a circuit, it is possible to determine its effect on the logical behavior of the circuit. There are several advantages in using logical fault models rather than physical fault models [Ref. 6]:

  - Once a logical fault model adequately reflects the physical failure modes of a circuit, the fault analysis becomes a logical problem.
  - It is possible to construct logical fault models that are applicable to many different technologies. In this case, fault analysis becomes relatively

technology-independent. The computer programs can then be written for fault simulation and test generation without losing their usefulness when changes in technology occur.

- By using a logical fault model, it is possible to perform a test for faults in cases when the effect on circuit behavior is not clear.
- A logical fault model often covers a large number of different physical faults, resulting in a substantial decrease in the complexity of fault analysis. There are five classes of logical faults [Ref. 7]:
  * Input stuck-at faults - When each of the input lines may be stuck-at-0 or stuck-at-1.
  * Output stuck-at faults - When any number of output lines may be stuck; each line may be stuck-at-0 or stuck-at-1.
  * Input bridgings - When any two input lines may be bridged. Bridging (short circuit) faults have become increasingly important for VLSI devices.
  * Output bridgings - All bridgings between any number of output lines.
  * Feedback bridgings - Bridgings between one input and one output line.

A commonly used fault model is the stuck-at model. A faulty gate input in a circuit is modeled as a stuck-at-0 (s-a-0) or stuck-at-1 (s-a-1). When a certain number of test vectors are applied to the CUT, the percentage of fault coverage depends on the number of s-a-0 or s-a-1 faults detected by the test patterns.

The stuck-at model does not take into account all possible defects, but acts as a global type of model. This model assumes that a logical gate input or output is fixed to either a logic 0 or a logic 1. For example, the pattern applied to the

fault-free AND gate in Figure 2.1 has an output of 0, since the A input is 0 on the A input and 1 on the B input. But the pattern in Figures 2.2 shows an output equal to 1, since the A input is perceived as a 1 even though a 0 is applied to that input. Therefore, the pattern shown in Figure 2.2 is a test for the A input (s-a-1), because the good machine responds differently from the faulty machine [Ref. 8]. Some techniques are available to reduce the complexity of fault simulation, but it is still a time consuming and expensive task. Testing of a sequential machine is more difficult than with a combinational machine as the output of a sequential machine depends on both the present input but also on the internal state of the machine.

One problem in Complementary Metal Oxide Silicon (CMOS) using this model is that a number of faults could change a combinational network into a sequential network. If this occurs, the combinational patterns are no longer effective in testing the circuit. As mentioned previously, single stuck-at fault test sets seem to provide acceptable levels of fault coverage for devices fabricated with current technology. The major problem in developing test sets for multiple fault detection is the large number of possible faults. For example, it is easy to verify that 10 nodes in a circuit may have 20 single stuck-at faults, 180 double faults, 960 triple faults, and 59,048 possible stuck-at patterns [Ref. 9]. For VLSI circuits containing in excess of 10,000 nodes, explicit test generation for anything other than single faults is impractical.

In summary, fault simulation in VLSI circuit verification has some difficulties. Increasing circuit complexity will increase the time consumption for the simulation of all gate-level faults. However, consideration of only single-stuck faults may be inadequate. Multiple faults, non-stuck faults and suspended temporarily at intervals type faults are important but difficult to investigate [Ref. 10].

A commonly used test pattern generation is the D-Algorithm, developed by Paul Roth [Ref. 11], is widely used to generate a test vector for a given fault.

This method is usually used with gate-level circuit models and stuck faults. This algorithm attempts to construct a sensitized path over which an error signal can propagate from the fault location to an observable primary output line. Using a backtracking approach based on the circuit structure, the D-Algorithm searches the space of possible test patterns for the given fault. This method, in its most general form, can always find a test vector that can sensitize a single fault. In the case where the fault is undetectable, it can prove that no test pattern exists. The D-Algorithm is particularly well-suited to test generation for circuits designed using Level Sensitive Scan Design (LSSD). A large number of practical test generation programs and algorithms are based on the D-Algorithm [Ref. 6]. In Goel and Rosales [Ref. 11], a new algorithm called PODEM is described in an attempt to reduce the backtracking of the D-Algorithm [Ref. 1]. In the PODEM algorithm, the path from an output node to a primary input is traced and branching decisions are made heuristically at each step of backtrack. After reaching the primary input, a simulator checks if the initial objective function has been achieved. If not, the algorithm will repeat until the target fault is sensitized. For moderately large circuits, PODEM is proved very effective. With the vast increase in circuit density, the ability to generate test patterns automatically and to conduct fault simulation with these patterns has decreased. Therefore, some manufacturers skip these difficult approaches and accept the risks of shipping a defective product. One approach, called Design for Testability, is used to reduce the cost of testing [Ref. 8].

## B. DESIGN FOR TESTABILITY

Design for testability (DFT) is motivated by the need to reduce the costs of testing. The main test considerations are difficulty of test generation, test sequence length, test application cost, fault coverage and fault resolution. The costs involve

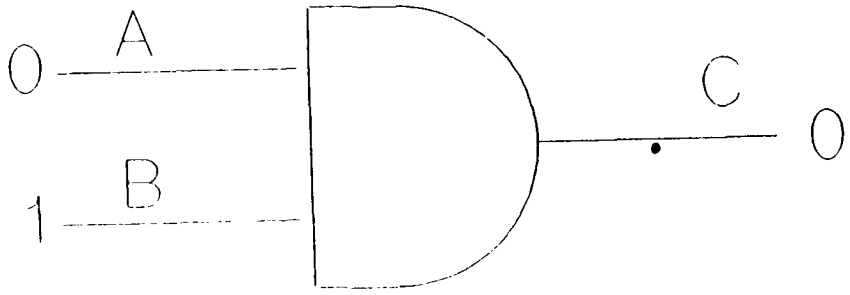



**Figure 2.1: Fault Free AND Gate**

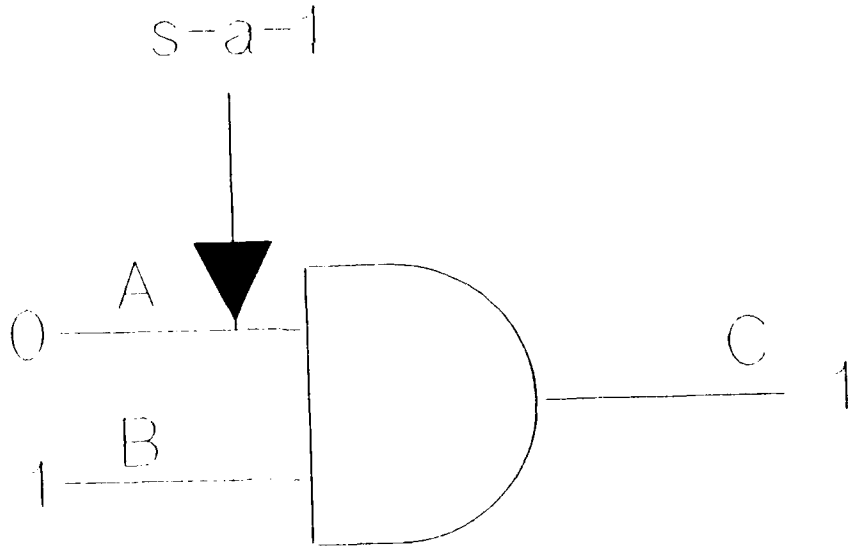



**Figure 2.2: Faulty AND Gate**

the computer time required to generation the test pattern, the personnel to write the test pattern program and to test the equipment [Ref. 6]. The DFT objective is to design circuits from the outset to limit in magnitude the test generation efforts and test verification [Ref. 12]. Testability relies on two concepts, controllability and observability. Controllability is the ability to set and reset every node internal to the circuit. Observability is the ability to observe (either directly or indirectly) the state of any node in the circuit.

Three basic approaches to DFT are ad-hoc testing, a structured design approach, and built-in self testing. For the purpose of this thesis, the built-in self testing is the method of concentration. Different techniques in built-in self testing are described below [Ref. 13].

- Concurrent Built-in Logic Block Observers (CBILBO)

  CBILBO offers many distinct features for both test application and response analysis. First, the test length is the shortest among several BIST methods studied. Thus, this scheme is suited to test large sequential machines. Second, as test generation and signature analysis are separated, the proposed approach generates exhaustive or pseudo-exhaustive tests for the circuit under test. Thus, no test pattern generation program is necessary. Third, fault simulation may not be needed. Fourth, the control sequence is very simple as opposed to other BIST methods. Finally, on-line checking is possible. The main disadvantage of this proposed method is that the hardware cost is higher than other methods.

- Built-in Logic Block Observation (BILBO)

  The BIST approach using BILBO is used to test sequential machines. In this method, the signature data collected from the previous tests must be used as

the pattern for the next test. BILBO has a slightly lower hardware cost than the CBILBO approach, however, it results in lower fault coverage. To obtain a higher fault coverage, a detailed circuit analysis or fault simulation would be needed.

- Scan BIST

  The scan BIST approach is suited to test circuits where a scan path is employed and is therefore recommended for scan-based designs, although the test length is longer than the CBILBO or BILBO approach.

- Checking Experiment

  The BIST approach using checking experiments have the longest testing time. Since the testing time increases exponentially with the number of internal states, it has comparable fault coverage and hardware overhead as other previously mentioned methods. These BIST schemes are not attractive to test sequential machines. However, function-dependent checking experiment methods remain attractive, as lower hardware overhead can be obtained as compared to other BIST methods.

BIST approaches have been applied either partially or fully to commercial microprocessors and gate arrays, although focusing on DC functional testing. Current BIST techniques have difficulty achieving high fault coverage for sequential circuits due to randomness of test patterns and aliasing probability, etc. The focus here is on the BILBO approach. This technique combines the Scan Path and LSSD concepts with Signature Analysis [Ref. 14]. The complete test generation and observation arrangement can be implemented as shown in Figure 2.3 [Ref. 12]. The BILBO register on the left is used as a pseudo-random sequence generator to generate test

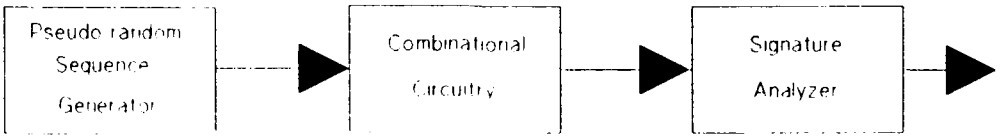



**Figure 2.3: The Complete Test Generation and Observation Arrangement**

patterns, which will be applied to the combinational circuitry. The BILBO register on the right is used as the signature analyzer. After a certain number of patterns are applied, the signature generated by the signature analyzer will be compared against the reference data [Refs. 8, 12]. In LSSD or other structure design techniques, a considerable volume of test data is required with the shifting in and out. Using the BILBO technique, the test data volume may be reduced by a factor of 100 for every 100 test patterns.

## C. SIGNATURE ANALYSIS

A test response compression method is used to solve the problem of analyzing and storing the large amount of data required for collecting responses. In this method, response data $R$ will be compacted to form $f(R)$, including the fault information.

The signature analysis technique was introduced in 1977 by Hewlett-Packard [Ref. 15]. This technique should be used when it is not feasible to compare test results with reference data. If the reference data is available at the same rate and synchronized with the data being tested, there is no advantage in using this technique. The key to signature analysis is to design a network which can excite

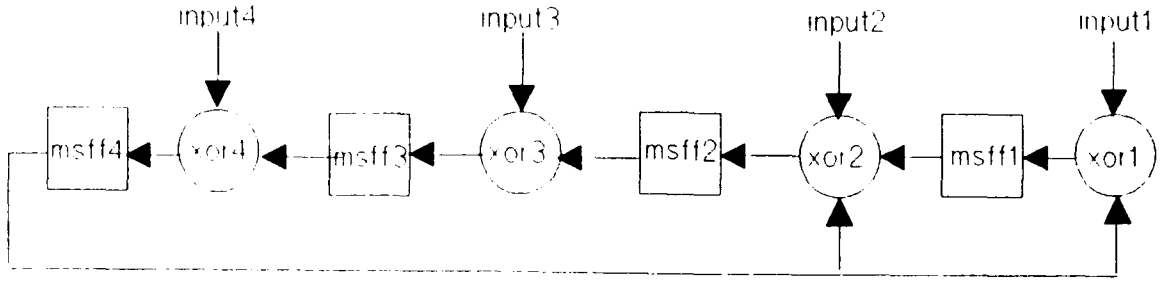



**Figure 2.4: MISR**

itself. For example, microprocessor based boards stimulate themselves using the intelligence of the processors driven by the memory on the board. The signature and its collection algorithm should meet four qualitative guidelines [Ref. 11]:

- The algorithm must be simple enough to be implemented as part of the built-in test network.
- The implementation must be fast enough to prevent it being a limiting factor during test time.
- In order to minimize the reference-signature storage volume, the algorithm must provide logarithmic compression of the output response data.
- The compression method must not lose information, especially if a fault is indicated by a wrong response from the CUT.

### 1. Signature Analysis

The integral part of the Signature Analysis is the Linear Feedback Shift Register (LFSR) or Multiple Input Signature Register (MISR) (See Figure 2.4). Basically, LFSR is built up of latch, with feedback connections. A polynomial of degree $n$ relates to an $n$-stage LFSR. The output of the last stage and some intermediate stages are fedback to the first stage via XOR-gates. The first and the last stage are at the rightmost and leftmost bit positions respectively, if the register shifts from right to left. The input from the CUT also feeds to the first stage via XOR-gate. In general, for any response data stream of length greater than 4, the probability of missing a faulty response when using an n-bit signature analyzer is the number of undetected errors divided by the number of total patterns, therefore,

$$\frac{2^{n-m}-1}{2^n-1} \approx 2^{-m} \tag{2.1}$$

where $m$ is the number of stages. As most practical circuits have many outputs, a MISR is used to form a signature of the CUT. The MISR circuit is currently considered to be the most efficient means of producing a signature of a multiple-bit data stream [Ref. 16]. The MISR is basically similar to LFSR, the main difference being the XOR placed between each stage in MISR and the output of the CUT. The output of CUT will feed into each stage via these XOR gates.

### 2. Advantage

The major advantage derived by this approach is that the number of bits that have to be compared to determine the correctness of the CUT is greatly reduced [Ref. 17].

### 3. Disadvantage

The major disadvantage of this approach is the difficulty to drive any information from the signature beyond a yes/no decision. It is difficult enough to investigate failure from the failing signature. Consequently, a failure in one node will cause failures to be observed at any other nodes [Ref. 18]. One problem when using signature analysis is that the volume of information contained in the signature is less than the volume of information contained in the actual data [Ref. 18]. Another problem using signature analysis is aliasing error, discussed in Chapter III.

# III. ALIASING IN MISR

Using the signature analysis techniques for testing circuits, considerable amount of attention has been paid to the aliasing problem: the result found in the signature analysis register is correct, however, the Circuit Under Test is in fact faulty. We restrict our discussion for combinational CUT.

Figure 3.1 shows an example of the experimental set-up for signature analysis. The CUT is supplied with a set of test patterns that is generated by a LFSR. The output of the CUT goes into the multiple input signature register (MISR) which is used as a compressor. Since the CUT is combinational, one does not have to deal with the initialization of the CUT. In addition, test patterns from LFSR are applied to the CUT synchronously. Therefore, the first pattern is applied to CUT after a certain time has passed such that the results are available at the output of the circuit and available at the input of the MISR. The MISR will change to a new value and wait for the next value from the CUT. Before starting a test, we must initialize the LFSR and MISR. When all the test patterns have been applied, the contents of the MISR form the signature that is compared to a reference value. Following this comparison, the determination is made whether the circuit response is faulty or not.

## 1. Aliasing Probability

The problem of aliasing is generally characterized as a probability. The most widely accepted measure is the probability of aliasing in MISR. It is generally assumed that the probability of error in each bit is $p$, i.e., equally likely, and all of the inputs are independent. Under this assumption, as developed in the previous chapter, the aliasing probability of an MISR of length $k$ is $P_{al} = \frac{1}{2^k}$. This result is

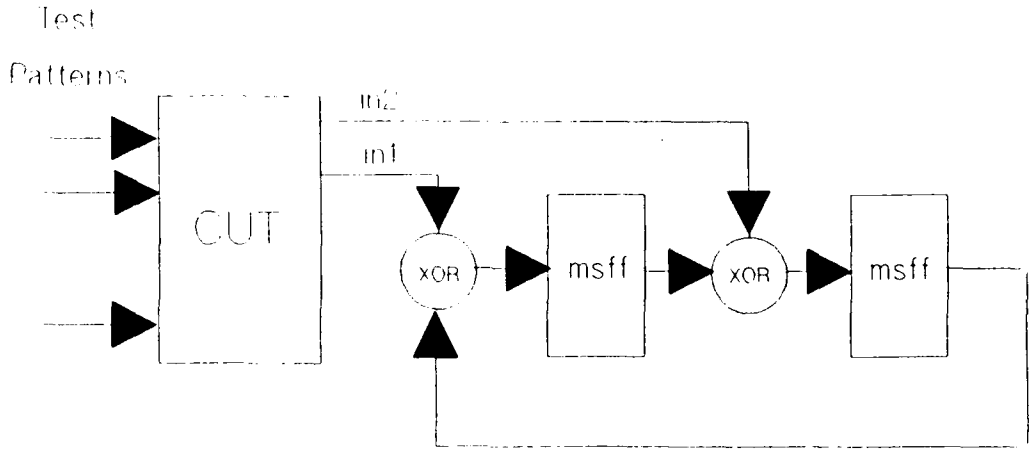



**Figure 3.1: Experimental Set-up for Signature Analysis**

independent of the effect of the fault on the output of the CUT. The probability of aliasing depends only on the number of stages of the MISR. Therefore, in order to reduce the probability of aliasing, the length of the MISR must be increased. For instance, adding one stage to the MISR reduces the probability of aliasing by a factor of 2. Unfortunately, the "equally likely" assumption on bit error probability seems unreasonable for a practical circuit [Ref. 19].

### 2. Notation and Definition

As mentioned above, the $k$ available outputs of the CUT are connected to the $k$ MISR inputs. If MISR has more inputs than CUT outputs, the remaining free MISR inputs will be assumed to be connected to a dummy CUT output [Ref. 20]. The operation of the MISR of length $k$ can be described in matrix notation. The state of the register can be represented by a column vector:

$$Y(t) = [y_1(t), y_2(t), \cdots, y_k(t)]^T . \tag{3.1}$$

and the CUT outputs can be represented by a column vector:

$$X(t) = [x_1(t), x_w(t), \cdots, x_k(t)]^T . \tag{3.2}$$

The next state of the register is calculated by multiplying the current state vector by the next state matrix, $C$, and adding the current input to the product. Note that the matrix C represents the MISR polynomial with $C_i$ as coefficients.

$$\begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ \vdots \\ y_k(t) \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & c_3 & \cdots & c_k \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \begin{pmatrix} y_1(t-1) \\ y_2(t-1) \\ \vdots \\ \vdots \\ y_k(t-1) \end{pmatrix} \oplus \begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ \vdots \\ x_k(t) \end{pmatrix} \tag{3.3}$$

Equation 3.3 can be rewritten as,

$$Y(t) = C \cdot Y(t-1) \oplus X(t) \tag{3.4}$$

The autonomous behavior of the register is described by Equation 3.4 by setting all input bits in Equation 3.2 to zero

$$x_1(t) = x_2(t) = \cdots = x_k(t) = 0 . \tag{3.5}$$

In the event of a fault in the CUT, the present output vector can always be represented as the linear superposition of the correct-circuit sequence,

$$G(t) = [g_1(t), g_2(t), \cdots, g_k(t)]^T \tag{3.6}$$

and an error sequence,

$$E(t) = [e_1(t), e_2(t), \cdots, e_k(t)]^T \tag{3.7}$$

therefore,

$$\begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ \vdots \\ x_k(t) \end{pmatrix} = \begin{pmatrix} g_1(t) \\ g_2(t) \\ \vdots \\ \vdots \\ g_k(t) \end{pmatrix} \oplus \begin{pmatrix} e_1(t) \\ e_2(t) \\ \vdots \\ \vdots \\ e_k(t) \end{pmatrix} \tag{3.8}$$

or

$$X(t) = G(t) \oplus E(t). \tag{3.9}$$

In case the fault is not detected in CUT, the

$$e_1(t) = e_2(t) = \cdots = e_k(t) = 0 \tag{3.10}$$

After $n$ clock cycles, the state of the MISR can be computed by applying iteratively Equations 3.4 and 3.9. It has been shown that $x(t)$ consists of bitwise modulo-2 sum of the fault free circuit outputs with the assumption that the register is in the initial state 0 [Ref. 20]. Consequently, an aliasing event occurs if the error sequence is applied and if there is at least one nonzero element in the error sequence to cause the MISR to return to an initial state.

### 3. Aliasing and Markov Process

William and Dahne [Ref. 19] stated that a Markov process is very similar to that of a sequential machine. This means that a Markov process can be defined in terms of state transitions. If the input sequence to the signature register is random, the transition is done with the probabilities of the respective input symbol. When this process occurs, the transition diagram for the Markov process has the same behavior as the operation of the signature register with random signals at the input is obtained (See Figure 3.2). Based on the facts above, the calculation is made that the probability of aliasing $P_{al}$ in the signature register of length $k$ is

$$P_{al} = \frac{1}{2^k} \tag{3.11}$$

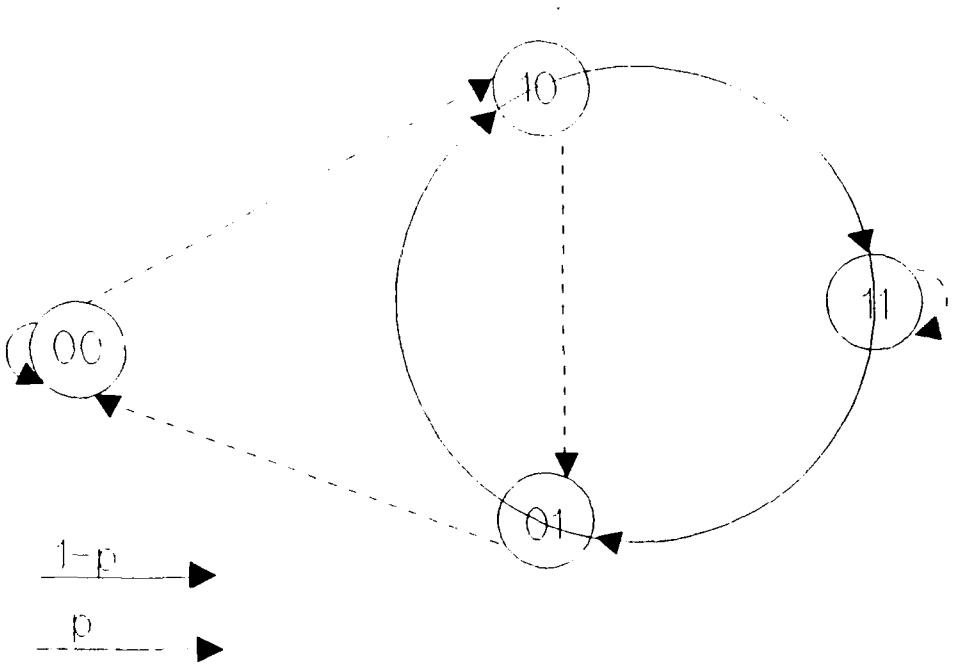



**Figure 3.2: Transition Diagram of the Markov Process for LFSR Implementing $x^2 + x + 1$.**

In the next chapter, we will investigate the aliasing probability by discussing the results of an experimental work.

# IV. A TEST SYSTEM ARCHITECTURE

## A. SYSTEM ARCHITECTURE

The purpose of this section is to investigate the aliasing probability in BIST by using MISR as a compressor. To illustrate the function, some circuits were connected as one overall chip. The Linear Feedback Shift Register (LFSR) is used as a test pattern generator, the Full Adder is used to represent Circuit Under Test (CUT) and finally, the Multiple Input Signature Analysis (MISR) is used as a compressor to reduce the cost of testing. We now explain each circuit in the next three subsections.

### 1. Linear Feedback Shift Register as a Test Pattern Generator

The most popular hardware pseudorandom sequence generator is the Linear Feedback Shift Register [Ref. 21]. The binary sequence at cell $i$ is generally considered to display attributes of pseudorandom binary sequence. The basic structure of LFSR consists of the shift register comprised of a number of stages, with feedback connections based upon the primitive polynomials used. Primitive polynomials are used as a base structure because they can generate maximum length sequences. For instance, a maximum length sequence generator with $n$-stages can deliver $2^n - 1$ distinct $n$ bit long patterns except the zero pattern. Since we have to connect outputs of LFSR to inputs of CUT, the number of stages of the LFSR depends on the number of inputs of CUT. Therefore, the degree of primitive polynomials used depends on the number of CUT input. For instance, a 4-bit adder requires 8 stages of LFSR, or primitive polynomials of a degree of 8. Some unique properties of LFSR that implements a primitive polynomial are listed below [Ref. 21]:

- Staggering Relationship among successive outputs

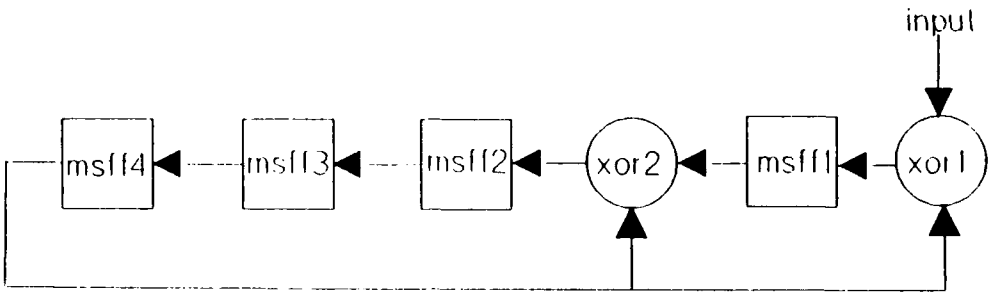



**Figure 4.1: LFSR Described by Polynomial $x^4 + x + 1$**

If the contents of the stages are used directly as pattern outputs, then the bit-streams obtainable from the stages will contain basically the same bit-sequence, but the bit pattern will be staggered by $i$ cycle in each stage relative to the one on its left.

- Cyclic Periodicity

  The pattern generated will continue through $2^n - 1$ cycles, and after that, the content of stages will become the starting pattern and the sequence will repeat periodically.

- Exclusion of all 0 pattern

  The LFSR should not be initialized with all 0, because in this case the feedback is always 0. The contents of the stage will remain 0 in all shift cycles and the LFSR can not generate any nonzero pattern.

- Superposition Effects

This is a very important property of LFSR which occurs from the influence of the individual seedbits in a LFSR. The sequence pattern generated by the LFSR using one initial value is seen to correspond, cycle for cycle, to the XOR (module-2-add) sum of the patterns in the other sequence pattern that uses two different initial values.

### 2. Full Adder as a Circuit Under Test

A Full Adder is a three input, two output combinational circuit that adds two 1 bit binary numbers. The Full Adder circuit is shown in Figure 4.2. To build an adder for 4 bit words, the full adder is replicated four times. In the experimental 4-bit Full Adder used, the carry out of one bit was used as the carry into its left neighbor. The carry into the right most bit is wired to 0. Adders that do not have this ripply carry delay, and hence are faster, also exist. An example of a 1 bit full-adder is shown in Figure 4.2. [Ref. 22]

### 3. Multiple Input Signature Register as a Compressor

The structure of MISR generated is basically similar to an LFSR. The only difference is that the input stream for each stage feeds through an XOR gate. Therefore, the number of stages is equal to the number of input XOR gates. In each test cycle, the output of each stage will be XORed with a bit of an input stream and shifted to the next stage to the left. The feedback configuration is based on the primitive polynomials used, and also must be XORed with an input stream. Two general structures can be considered for implementing MISR: external XOR and internal XOR [Ref. 20]. We are concerned in this thesis only with the internal OR structure due to its economical implementation [Ref. 23]. As in LFSR, the structure of MISR also depends on the output of the CUT. For instance, using a CUT with $k$ outputs, a $k$ stage MISR is needed. Therefore, primitive polynomials of degree $k$

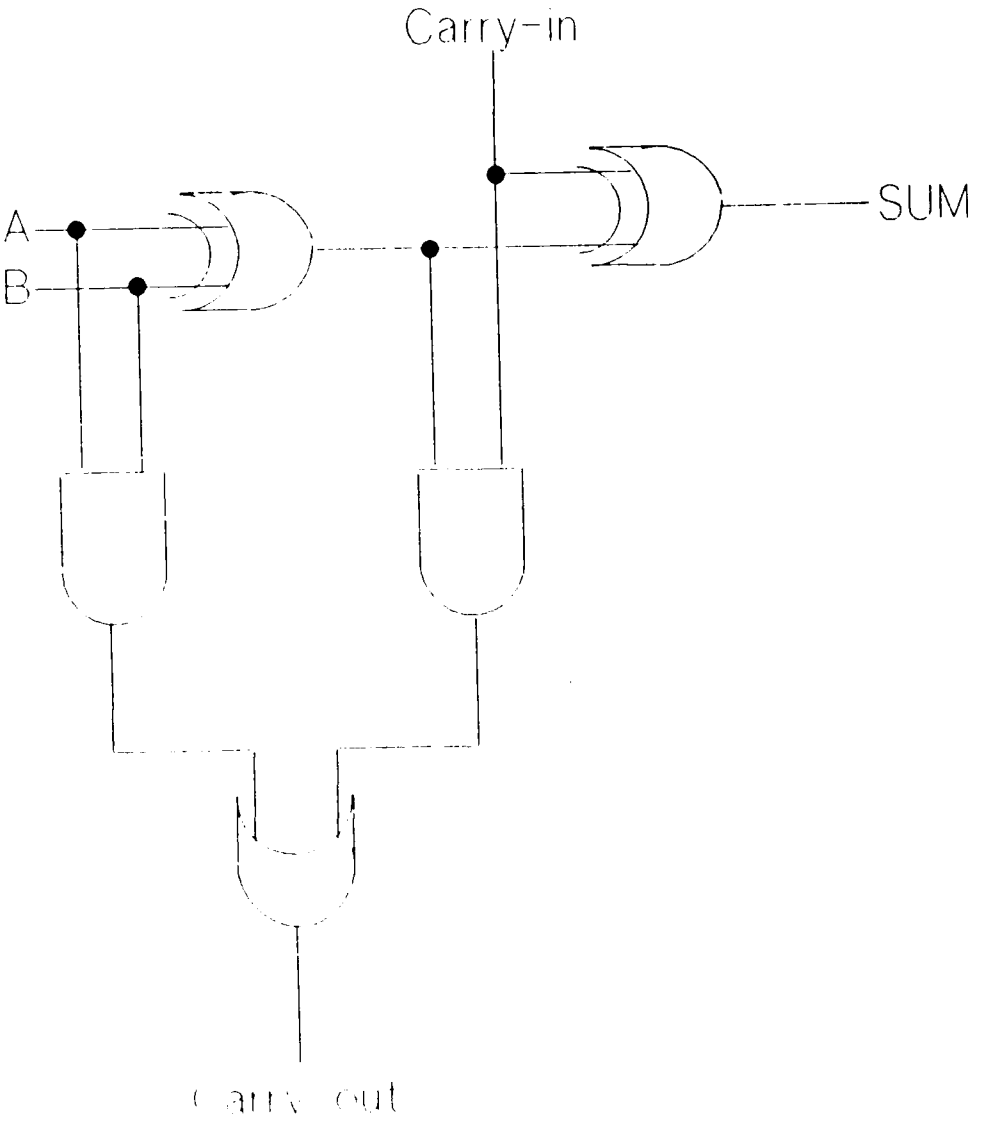



**Figure 4.2: Full Adder**

are required. If the CUT has fewer outputs than $k$, the remaining free MISR inputs will be assumed to be connected to dummy CUT outputs, i.e., constantly 0 [Ref. 4]. Both the state of the MISR and the CUT output can then be represented by vectors of $k$ bits.

## B. GENERATOR

Software provided by NW Laboratory for Integrated Systems [Ref. 24] was used to simulate the network. The network description file for LFSR, MISR, and Full Adder (CUT) was created by using subroutines written in "C". The program listing may be found in Appendix A. Three generators were created:

- GLFSR - to generate a network description file for Linear Feedback Register for a given primitive polynomial [Ref. 23].
- GADD - to generate a network description file for a ripple carry adder.
- GMISR - to generate a network description file for Multiple Input Signature Register for any primitive polynomials.

The network description file may be found in Appendix B.

## C. TEST CIRCUIT

Finally, the GTEST combines LFSR, adder, and MISR into one module and generates a connection description file. Besides, the GTEST can simulate the entire module. CAD tools used in GTEST include the following:

- NETLIST - creates a flattened netlist representation from an hierarchical representation;
- PRESIM - creates a binary circuit representation from the flat circuit representation;

- GEN_TIME - converts a simple timing file in a RNL compatible format;
- RNL - Runs the simulation.

The pattern generator GLFSR, the compressor GMISR, and the CUT will be combined to form a network description file created by GLFSR, GMISR and GADD. This description file is converted to an intermediate file by using NETLIST command. Another conversion occurs when producing a binary file suitable for use in RNL and is done by the PRESIM command. RNL requires a file that contains a sequence of RNL commands. At the very least, this command file should load one or more libraries of standard functions and should read in the binary description of the circuit created by PRESIM. The command file should also contain command and definitions of LISP function [Ref. 24].

## D. EXPERIMENTAL SET-UP

To demonstrate BIST circuitry, a 4 bit full-adder is used as a Circuit Under Test (CUT). As each adder has two inputs, an 8 stage LFSR is needed. This would feed 8 outputs into the CUT. The primitive polynomial used to generate the LFSR is $x^8 + x^6 + x^5 + x^3 + 1$. As a 4 bit full-adder has 5 outputs, 4 sums and 1 carryout, a 5 stage MISR is needed. The MISR can be described by the primitive polynomial $x^5 + x^2 + 1$. Examples for 2 and 8 bit full-adders will also be shown. Figure 4.3 shows the experimental set-up.

Before starting the experiment, an initial value for the LFSR must be given. The LFSR generates test patterns that feed into the CUT. Outputs of the good CUT feed into the MISR and generate signatures to be used as reference data. Next, we create a faulty CUT with stuck-at-0 by connecting one node in the last bit of the full-adder to the GND. Figure 4.4 depicts flowcharts of the generation of LFSR, Adder and MISR, and how all circuits are combined into one module.

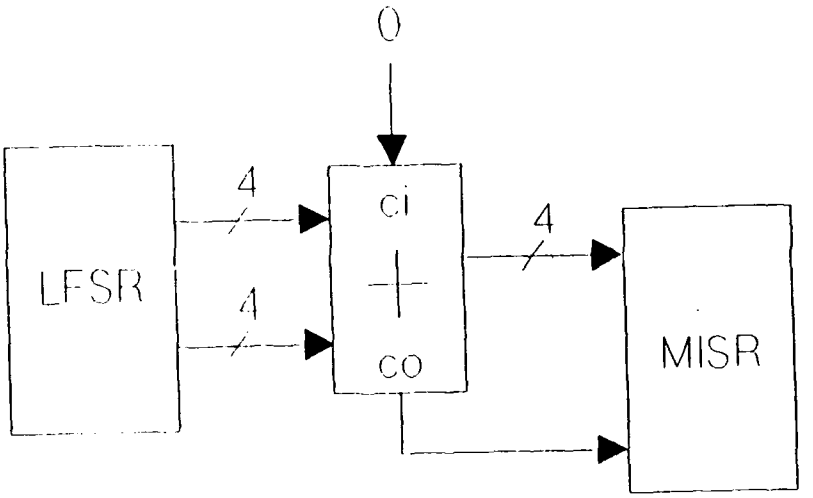



**Figure 4.3: Experimental Set-up**

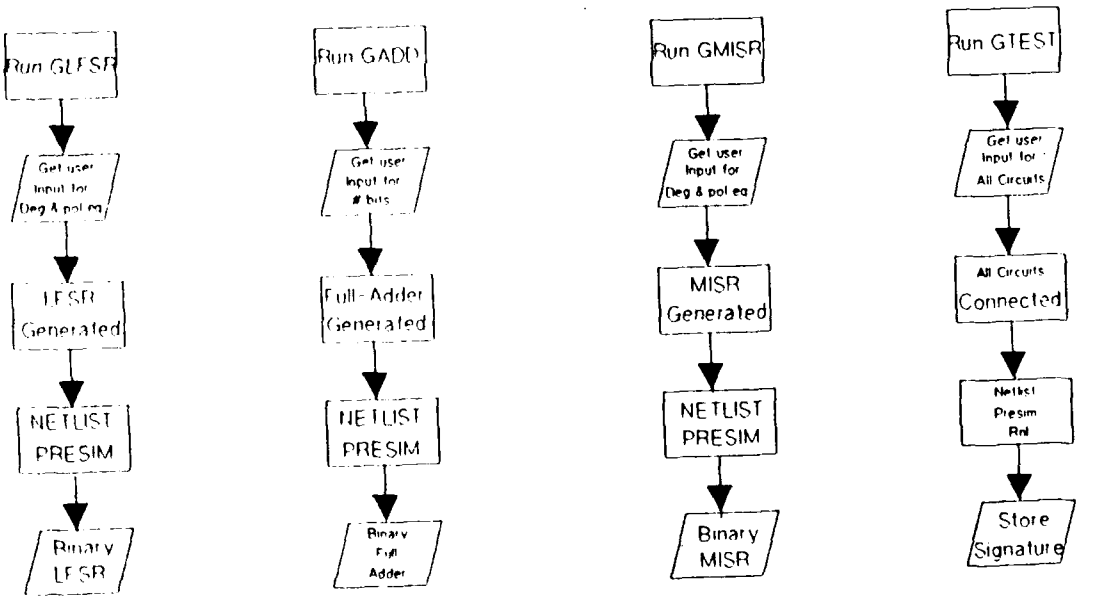



**Figure 4.4: The Flow to Generate LFSR, Full-Adder, MISR and the Combination of all Circuits into One Module**

The derived signature will be compared with reference data to investigate the possibility of aliasing. To observe aliasing in different fault locations, the node is connected with Vdd, named stuck-at-1, and the signature compared with the reference data. The experiment repeats with a different error by changing the connection of the node. An example of this is to connect the node of the first bit full-adder with GND and Vdd, respectively.

## E. EXPERIMENTAL EVALUATION

To investigate the probability of aliasing, a set of simulations were performed on three different CUTs with different polynomials of LFSR and MISR. The first experiment uses a 2 bit full-adder as the CUT. In this experiment, the LFSR is described by the polynomial $x^4+x+1$, and the MISR is described by the polynomial, $x^3 + x + 1$. Table 4.1 shows the aliasing probability for s-a-0 and s-a-1 faults with different initial states of LFSR. The aliasing probability is calculated by dividing the undetected occurrances with the total number of test vectors applied to the CUT. In our experiments, we always use $2^n - 1$ test vectors for an $n$-input adder since the largest adder under test is 8 bits, i.e., with a total of 16 inputs. The location of the stuck-at fault is at the input of the OR gate as shown in Figure 4.5. This OR gate is the LSB of the adder. The value for the SUM is equal in the cases of s-a-0, s-a-1, and fault free circuits. The only difference is the value of the carry-out for the s-a-1 circuit, which is less than the carry-out value of s-a-0 and fault free circuits. Therefore, a circuit with a s-a-0 fault will produce test patterns almost equal to that of the fault free circuit but causes aliasing of a s-a-0 fault to be greater than in a s-a-1 fault. Table 4.1 also shows that the possibility of aliasing is the same for different initial states and that aliasing probability in an s-a-0 fault is greater than with a s-a-1 fault. The second experiment uses a 4 bit full-adder as CUT, with the

**TABLE 4.1: ALIASING PROBABILITY WITH DIFFERENT INITIAL STATES OF LFSR ON 2-BIT ADDER**

| Initial State | Aliasing s-a-0 | Aliasing s-a-1 |
|---|---|---|
| 1010 | 0.8667 | 0.4667 |
| 1011 | 0.8667 | 0.4667 |
| 1101 | 0.9373 | 0.3333 |
| 1001 | 0.9333 | 0.3333 |
| 1111 | 0.8667 | 0.5333 |

LFSR described by the polynomial $x^8 + x^6 + x^5 + x^3 + 1$, and the MISR described by the polynomial, $x^5 + x^2 + 1$. An interesting case of aliasing was found in this experiment that showed the potential for the aliasing probability to be 1. Figure 4.6 shows that $P_{al} = 1$ for s-a-0 fault and for s-a-1 $P_{al} = 0.01667$ (Figure 4.7), where the position of the stuck-at fault is at the input of the XOR gate in LSB that produces SUM. Table 4.2 shows the expected value of SUM for good CUT using two different values of carry-in. Table 4.3 shows the value of SUM for two different faults.

By comparing Tables 4.2 and 4.3, the result shows that if the carry-in in the good circuit equals 0, the SUM will be equal for both the good and bad CUT. However, if the carry-in in the good CUT equals 1, then the aliasing probability in s-a-1 fault equals 1. This is shown in Figure 4.8. For s-a-0, $P_{al} = 0.0333$, as shown in Figure 4.9. In the third experiment observed, the position of the s-a-fault was

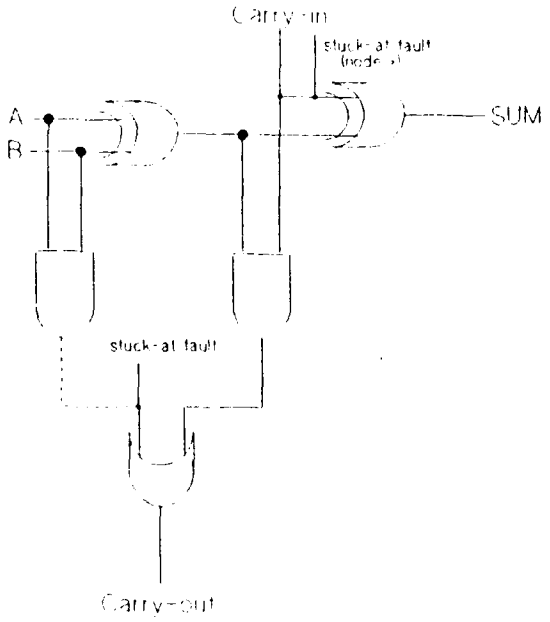



**Figure 4.5: One Bit Full Adder with s-a-fault**

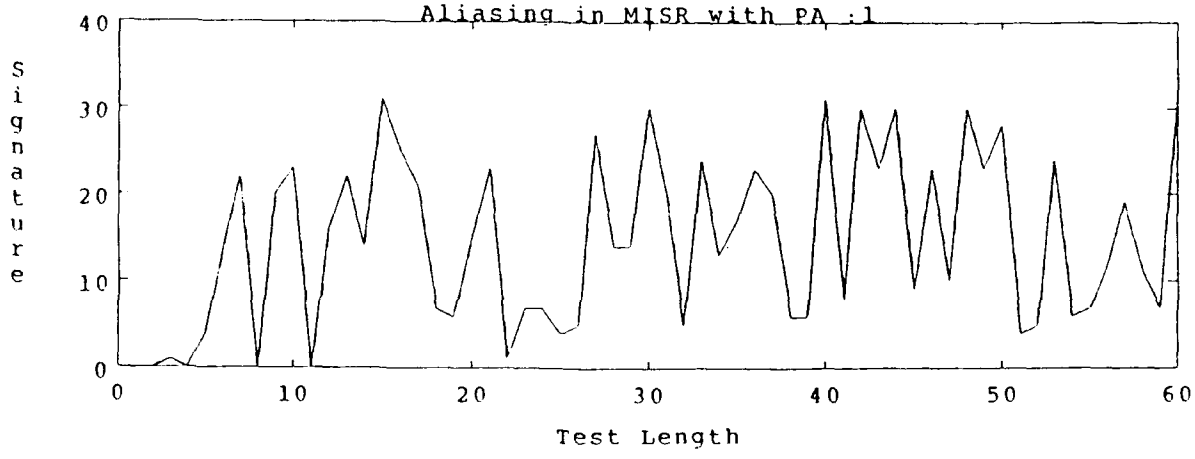



**Figure 4.6: The Aliasing Probability in s-a-0 Fault**

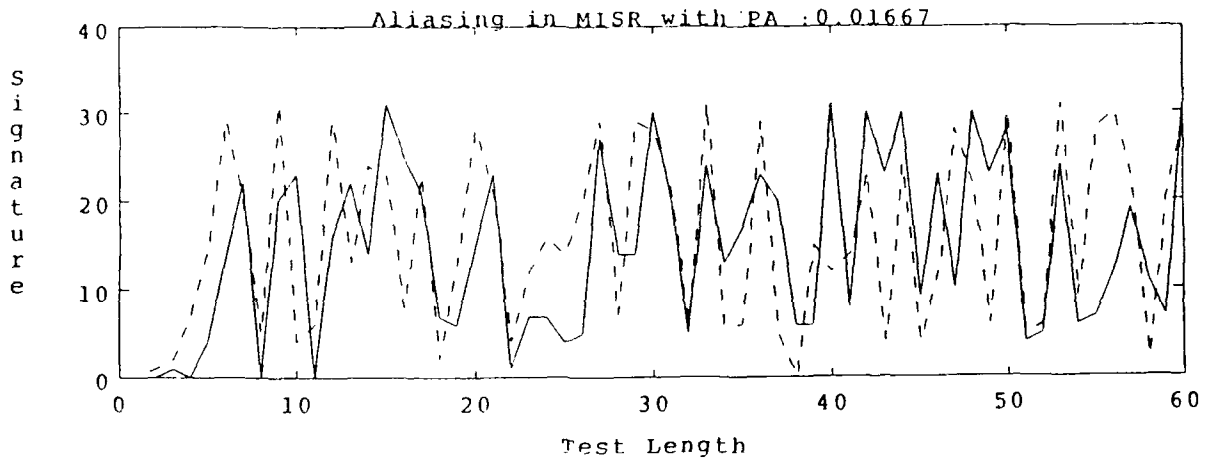



Figure 4.7: The Aliasing Probability in s-a-1 Fault

TABLE 4.2: EXPECTED VALUE OF SUM FOR GOOD CUT

| A | B | Carry in | SUM | Carry in | SUM |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

TABLE 4.3: THE VALUE OF SUM FOR DIFFERENT S-A-FAULTS

| A | B | Node x s-a-0 | SUM | Node x s-a-1 | SUM |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

changed to the input of AND gate. Table 4.4 shows the aliasing in s-a-0 and s-a-1 faults for different initial states of LFSR.

Table 4.4 demonstrates that the actual aliasing probabilities are greater than the aliasing probability calculated given by the formula in Section 3.3. For example, if a 5 stage MISR is used, then the aliasing probability is calculated as $P_{al} = \frac{1}{2^5} = 0.03125$ instead of 0.0333. As in previous experiments, the aliasing in s-a-1 faults is less than aliasing in s-a-0 faults.

The third experiment uses an 8 bit full-adder as CUT. The LFSR is described by the polynomial $x^{16}+x^5+x^3+x^2+1$ and the MISR is described by the polynomial $x^9 + x^4 + 1$. Table 4.5 shows the aliasing in s-a-0 and s-a-1 faults for four different initial states of LFSR.

Table 4.5 shows that almost all of the probability of aliasing, either in the s-a-0 fault or in the s-a-1 fault, is equal to 0. These results are close to the aliasing probability as calculated by the formula. For example, with a 9 stage MISR, the

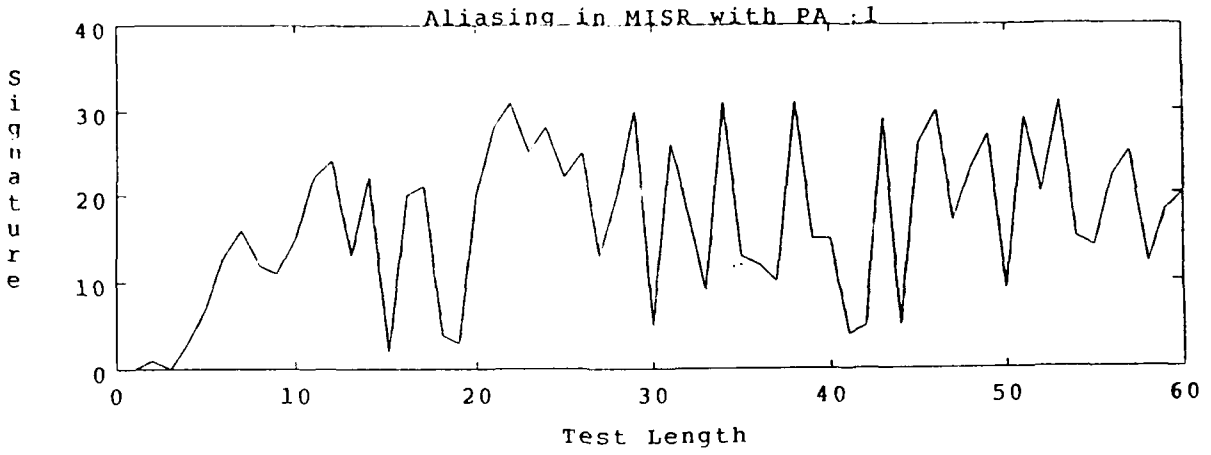



**Figure 4.8: The Aliasing Probability in s-a-1 Fault Using 5 Stages MISR**

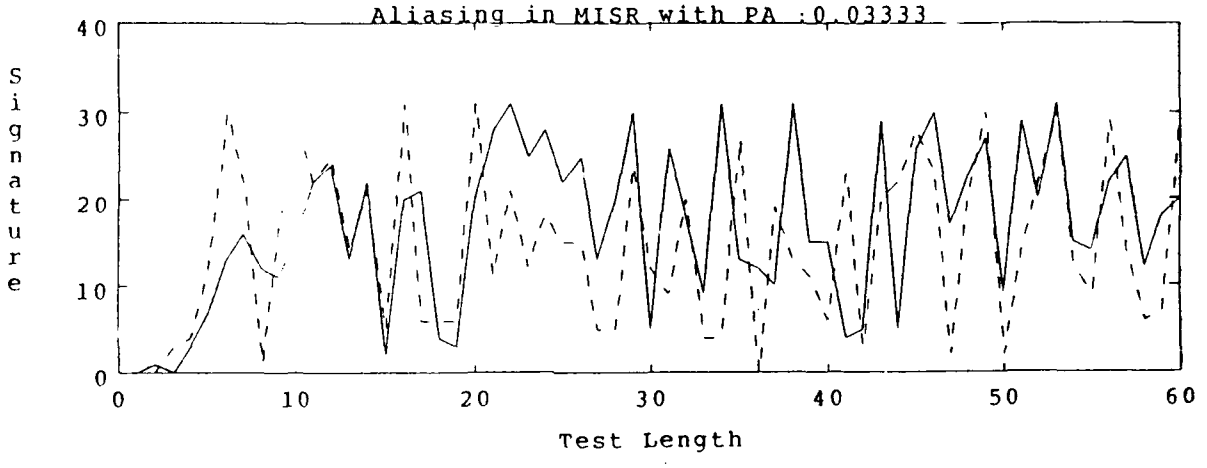



**Figure 4.9: The Aliasing Probability in s-a-0 Faults Using 5 Stages MISR**

TABLE 4.4: ALIASING PROBABILITY IN 5 STAGE MISR WITH DIFFERENT INITIAL STATES

| INITIAL STATE | ALIASING s-a-0 | ALIASING s-a-1 |
|---|---|---|
| 10110101 | 0.3608 | 0.1373 |
| 10001010 | 0.3176 | 0.1294 |
| 11100111 | 0.2157 | 0.1529 |
| 11111111 | 0.2510 | 0.1098 |

TABLE 4.5: ALIASING PROBABILITY IN 9 STAGE MISR WITH DIFFERENT INITIAL STATES

| INITIAL STATE | ALIASING s-a-0 | ALIASING s-a-1 |
|---|---|---|
| 0110011110000111 | 0.0000 | 0.0000 |
| 1110101011111111 | 0.0039 | 0.0000 |
| 1000101000000000 | 0.0039 | 0.0000 |
| 1111111111111111 | 0.0000 | 0.0000 |

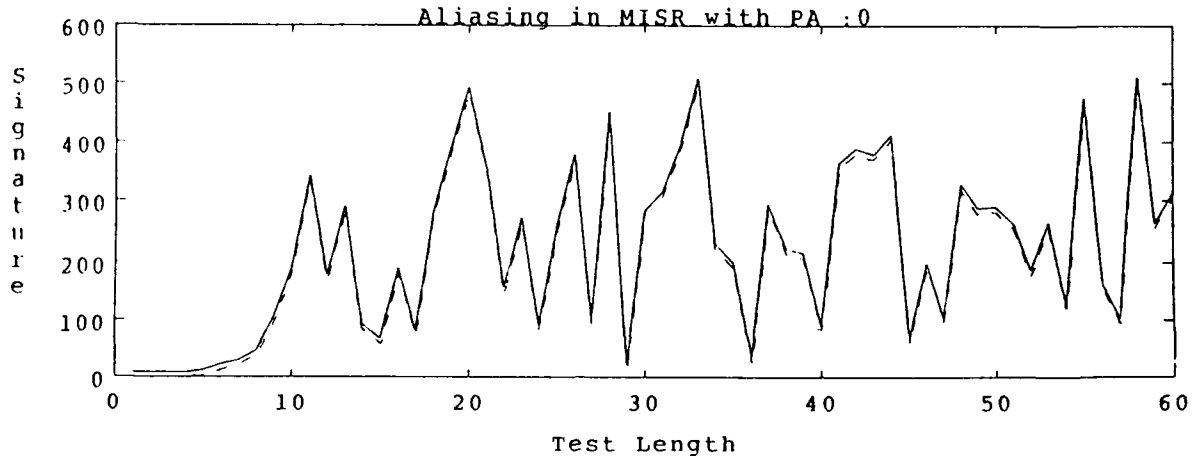



**Figure 4.10: Aliasing Probability in 9 Stages MISR**

aliasing probability is $2^{-9}$ or 0.0019. Figure 4.10 shows a similar signature for good and bad CUT. The only difference is that the good circuit has higher pattern values than the circuit with the s-a-0 fault. By observing the aliasing in cases described above, it is demonstrated that the aliasing in s-a-1 faults is smaller than aliasing in s-a-0 faults. Also shown is that with increasing numbers of stages of MISR, the aliasing probability will decrease as stated [Ref. 19].

# V. CONCLUSIONS

The testing of an integrated circuit chip is as important as its design. If the chip designed is to be used in a network, it is essential to verifty that it performs the intended function. This testing requirement must be considered during the design phase. One approach used in this thesis is the Built-In Self Test (BIST), where a Linear Feedback Shift Register (LFSR) is used as a pseudorandom sequence generator. An LFSR may generate $2^n - 1$ test patterns into a circuit under test (CUT).

Several experiments were performed using a full adder as the cut with a MISR as a compressor. The set of programs were written to stimulate the LFSR, MISR and full-adder (CUT) for different polynomials according to the width of the adder. Several experiments have been performed to investigate the aliasing probability in MISR by using different polynomials of LFSR, MISR, and different sizes of full-adder. By using the Markov process approach, it was proven that the probability of aliasing in $n$ stage of MISR is approximated by $2^{-n}$ [Ref. 19] . The aliasing observed in the experimentation process closely replicated the formula. In order to get accurate aliasing probability, an experiment having length $2^m - 1$ must be tested, where $m$ is the number of stages of LFSR. For example, using 16 stages of LFSR required a 65535 test length to use all the test patterns generated by the LFSR. By observing all of the aliasing probabilities for different circuits, the experiments allowed us to conclude that the aliasing probability in s-a-1 fault is not always the same as that in a s-a-0 fault. A second conclusion is that if the number of stages of MISR is increased, the aliasing will be reduced as expected and the value of aliasing in either s-a-0 or s-a-1 is greater than the aliasing probability calculated.

# APPENDIX A
## Programs For Thesis Project

### GADD to Generate Full-Adder

```
#include <stdio.h>
#define fname_len 20
#define len 200
#define MAXDEG 300
#define newline fprintf(fid, "\n")
main(argc, argv)
int argc;
char **argv;
{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len];
char fc[fname_len];
char fs[fname_len];
int i, j, k=1, N=0, T;
int xor[MAXDEG];
int max[16];
printf("\n\tEnter file name            : ");
scanf("%s",filebase);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);
printf("\n\tEnter number of bits       : ");
scanf("%d",&N);
/*
-------------------------------
CREATE NETWORK DESCRIPTION FILE.
-------------------------------
*/
strcat(filebase,".net");
fid= fopen(filebase,"w");
fprintf(fid, "(load \"lib.net\")\n");
fprintf(fid, "(load \"xor2.net\")\n");
newline;
fprintf(fid, "(");
fprintf(fid, "node"); newline;
```

```
for(i=N; i>0; i--)
        {
        fprintf(fid, "a%d b%d carin%d carout%d sum%d",i,i,i,i,i);
        newline;
        }
newline;
for(i=N; i>0; i--)
        {
        fprintf(fid, "invin%d1 invin%d2 invin%d3",i,i,i); newline;
        }
        newline;
for(i=N; i>0; i--)
        {
        fprintf(fid, "invout%d1 invout%d2 invout%d3",i,i,i); newline;
        }
newline;
for(j=N; j>0; j--)
   {
    fprintf(fid, "norin%d1 norin%d2",j,j);
    newline;
   }
newline;
for(i=N; i>0; i--)
        {
        fprintf(fid, "norout%d ",i);
        newline;
        }
newline;
for(j=N; j>0; j--)
   {
    fprintf(fid, "xoro%d1 xoro%d2",j,j);
    newline;
   }
newline;
for(j=N; j>0; j--)
   {
    fprintf(fid, "xori%d11 xori%d12 xori%d21 xori%d22",j,j,j,j);
    newline;
   }
newline;
for(j=N; j>0; j--)
   {
    fprintf(fid, "nandout%d1 nandout%d2",j,j);
    newline;
   }
```

```
newline;
for(j=N; j>0; j--)
   {
    fprintf(fid, "nandin%d11 nandin%d12 nandin%d21 nandin%d22",j,j,j,j);
    newline;
   }
newline;
fprintf(fid, ") "); newline; newline;
    for(i=N; i>0; i--)
    {
    fprintf(fid, "(xor2 xoro%d1 xori%d11 xori%d12 )\n",i,i,i);
    fprintf(fid, "(xor2 xoro%d2 xori%d21 xori%d22 )\n",i,i,i);
    newline;
    }
    newline;
    for(i=N; i>0; i--)
    {
    fprintf(fid, "(cnand nandout%d1 nandin%d11 nandin%d12 )\n",i,i,i);
    fprintf(fid, "(cnand nandout%d2 nandin%d21 nandin%d22 )\n",i,i,i);
    newline;
    }
    newline;
    for(i=N; i>0; i--)
    {
    fprintf(fid, "(nor norout%d norin%d1 norin%d2 )\n",i,i,i);
    }
    newline;
    for(i=N; i>0; i--)
    {
    fprintf(fid, "(cinvert invout%d1 invin%d1)\n",i,i);
    fprintf(fid, "(cinvert invout%d2 invin%d2)\n",i,i);
    fprintf(fid, "(cinvert invout%d3 invin%d3)\n",i,i);
    newline;
    }
    newline;
         for (i=N; i>0; i--)
         {
          fprintf(fid, "(connect a%d xori%d11)",i,i); newline;
          fprintf(fid, "(connect b%d xori%d12)",i,i); newline;
          fprintf(fid, "(connect carin%d xori%d21)",i,i); newline;
          fprintf(fid, "(connect xoro%d1 xori%d22)",i,i); newline;
          fprintf(fid, "(connect xoro%d2 sum%d)",i,i); newline;
          newline;
         }
          newline;
```

```
        for (i=N; i>0; i--)
        {
         fprintf(fid, "(connect a%d nandin%d11)",i,i); newline;
         fprintf(fid, "(connect b%d nandin%d12)",i,i); newline;
         fprintf(fid, "(connect nandout%d1 invin%d1)",i,i); newline;
         newline;
        }
         newline;
        for (i=N; i>0; i--)
         {
         fprintf(fid, "(connect xoro%d1 nandin%d21)",i,i); newline;
         fprintf(fid, "(connect carin%d nandin%d22)",i,i); newline;
         newline;
         fprintf(fid, "(connect nandout%d2 invin%d2)",i,i); newline;
         }
          newline;
        for (i=N; i>0; i--)
         {
          fprintf(fid, "(connect invout%d1 norin%d1)",i,i); newline;
          fprintf(fid, "(connect invout%d2 norin%d2)",i,i); newline;
         }
          newline;
        for (i=N; i>0; i--)
         {
         fprintf(fid, "(connect norout%d invin%d3)",i,i); newline:
         fprintf(fid, "(connect invout%d3 carout%d)",i,i); newline;
         newline;
         }
         newline;
        for (i=N; i>=2; i--)
         {
         fprintf(fid, "(connect carin%d carout%d)",i,i-1); newline;
         }
fclose(fid);
/*
-------------------------------------------------------------
CREATE A FLATTENED NETLIST REPRESENTATION AND A BINARY CIRCUIT
REPRESENTATION.
-------------------------------------------------------------
*/
sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);
}
```

## GMISR to Generate MISR

```
/*
-------------------------------------------------
PROGRAM NAME    : GMISR
INPUT           : Polynomial
OUTPUT          : Multiple Input Signature Register
APPLICATION     : To generate and simulate misr
                  for any polynomial.
-------------------------------------------------
*/
#include <stdio.h>
#define fname_len 20
#define len 200
#define MAXDEG 300
#define newline fprintf(fid, "\n")
main(argc, argv)
int argc;
char **argv;
{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len];
char fc[fname_len];
char fs[fname_len];
int i, j, N=0, T;
int xor[MAXDEG];
int max[16];
int M;
int c0 = 1, c1 = 1;
int c2 = 1, c3 = 1;
int c4 = 1, c5 = 1;
int c6 = 1, c7 = 2;
int c8 = 1, c9 = 1;
int k  = 1, m  = 1;
printf("\n\tEnter file name            : ");
scanf("%s",filebase);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);
printf("\n\tEnter degree of polynomial : ");
scanf("%d",&N);
M = N;
/*
--------------------------------
CREATE NETWORK DESCRIPTION FILE.
--------------------------------
```

```
 */
 strcat(filebase,".net");
 fid= fopen(filebase,"w");
 fprintf(fid, "(load \"lib.net\")\n");
 fprintf(fid, "(load \"xor2.net\")\n");
 fprintf(fid, "(load \"xor3.net\")\n");
 newline;
 fprintf(fid, "("); newline;
 fprintf(fid, "node r cl ");
 newline; newline;
 for(i=1; i<=N; i++) {
                      fprintf(fid, " ffo%d ",i);
                      newline;
                     }
 newline;
 for(i=1; i<=N; i++) {
                      fprintf(fid, " ffi%d ",i);
                      newline;
                     }
 newline;
 for(i=1; i<=N; i++) {
                      fprintf(fid, " in%d ",i);
                      newline;
                     }
 newline;
 while (N == 0);
 max[N] = 1;
 max[0] = 1;
    for (i = N-1; i>0; i--)
     {
      printf("\n\tX^%d Enter coef. 1 or 0     : ",i);
      scanf("%d",&max[i]);
       if (max[i] != 1)
           max[i]=0;
     }
      printf(" \n\n\tX^%d + ",N);
      for (i=N-1; i>0; i--)
        {
         if (max[i] == 1)
          printf("X^%d + ",i);
        }
          printf("1\n");
      for (i=N-1; i>0; i--)
        {
if (max[i] == 1)
```

```
   {
    k++;
   }
else if (max[i] == 0)
   {
     m++;
   }
      }
for(j=1; j<=m; j++)
   {
    fprintf(fid, "xorout2%d xorin2%d1 xorin2%d2 ",j,j,j);
    newline;
   }
newline;
for(i=1; i<=k-1; i++)
   {
    fprintf(fid, "xorout3%d xorin3%d1 xorin3%d2 xorin3%d3 ",i,i,i,i);
    newline;
   }
fprintf(fid, ") "); newline; newline;
for(i=1; i<=N; i++) fprintf(fid, "(msff ffo%d ffi%d c1)\n",i, i);
newline;
    for(i=1; i<=m; i++)
    {
    fprintf(fid, "(xor2 xorout2%d xorin2%d1 xorin2%d2 )\n",i,i,i);
    }
    newline;
    for(i=1; i<=k-1; i++)
    {
    fprintf(fid, "(xor3 xorout3%d xorin3%d1 xorin3%d2 xorin3%d3)\n",i,i,i,i);
    }
    newline;
         for (i=N-1; i>0; i--)
         {
         if (max[i] == 1)
           {
          fprintf(fid, "(connect ffi%d xorout3%d)",N,c2); newline;
          c2++;
          fprintf(fid, "(connect xorin3%d1 ffo%d)",c8,N-1); newline;
          fprintf(fid, "(connect xorin3%d2 ffo%d)",c8,M); newline;
          fprintf(fid, "(connect xorin3%d3 in%d)",c8,N); newline;
          newline;
          c8++;
          N--;
           }
```

```
        else if (max[i] == 0)
          {
               fprintf(fid, "(connect ffi%d xorout2%d)",N,c6); newline;
               c6++;
               fprintf(fid, "(connect xorin2%d1 ffo%d)",c9,N-1); newline;
               fprintf(fid, "(connect xorin2%d2 in%d)",c9,N); newline;
               newline;
               N--;
               c9++;
          }
         }
        fprintf(fid, "(connect ffi%d xorout2%d)",c4,c6); newline;
        fprintf(fid, "(connect xorin2%d1 ffo%d)",c9,M); newline;
        fprintf(fid, "(connect xorin2%d2 in%d)",c9,N); newline;
fclose(fid);
/*
-------------------------------------------------------------
CREATE A FLATTENED NETLIST REPRESENTATION AND BINARY CIRCUIT
REPRESENTATION.
-------------------------------------------------------------
*/
sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);
}
```

## GTEST to Combine All Circuits

```
/*
-----------------------------------------------------
PROGRAM NAME    : GTEST
INPUT           : LFSR, CUT, and MISR.
APLLICATION     : To combine all circuits and simulate.
-----------------------------------------------------
*/
#include <stdio.h>
#include <math.h>
#define fname_len 20
#define i_len 60
#define len 200
#define MAXDEG 300
#define newline fprintf(fid, "\n")
main(argc, argv)
int argc;
char **argv;
{
FILE *fid;
char filebase[fname_len], memory[40];
char fb[fname_len], fc[fname_len], fs[fname_len];
char fa[fname_len], fm[fname_len], fl[fname_len];
int i, j, k=1, N=0, M, T,l=0, P, R=0, r=1, rr=0;
int xor[MAXDEG];
int max[16];
printf("\n\tEnter file name for lfsr                : ");
scanf("%s",fl);
printf("\n\tEnter file name for full adder          : ");
scanf("%s",fa);
printf("\n\tEnter file name for misr                : ");
scanf("%s",fm);
printf("\n\tEnter file name for bist                : ");
scanf("%s",filebase);
strcpy(fb,filebase);
strcpy(fs,filebase);
strcpy(fc,filebase);
printf("\n\tEnter number of bits                    : ");
scanf("%d",&N);
R  = 2*N;
for(i=1; i<=R; i++)
{
r = 2*r;
}
r = r-1;
rr = 2*r;
```

```
M = N+1;
T = N;
P = N;
/*
--------------------------------
CREATE NETWORK DESCRIPTION FILE.
--------------------------------
*/
strcat(filebase,".net");
fid= fopen(filebase,"w");
fprintf(fid, "(load \"lib.net\")\n");
fprintf(fid, "(load \"%s.net\")\n",fa);
fprintf(fid, "(load \"%s.net\")\n",fm);
fprintf(fid, "(load \"%s.net\")\n",fl);
newline;
         for (i=2*N; i>0; i=i-2)
          {
           fprintf(fid, "(connect ffout%d a%d)",i,i/2); newline;
          }
           newline;
         for (i=2*N-1; i>0; i=i-2)
          {
           fprintf(fid, "(connect ffout%d b%d)",i,N); newline;
           N--;
          }
           newline;
           fprintf(fid, "(connect carout%d in%d)",T,M); newline;
         for (i=M-1; i>0; i--)
          {
           fprintf(fid, "(connect sum%d in%d)",i,i); newline;
          }
           newline;
fclose(fid);
/*
----------------------------------------
CREATE A FLATTENED NETLIST REPRESENTAION.
----------------------------------------
*/
sprintf(memory,"netlist %s.net  %s.sim -tcmos-pw",fb,fb);
system(memory);
sprintf(memory,"presim %s.sim %s",fb,fb);
system(memory);
/*
-----------------------------------------------------
CONVERT A SIMPLE TIMING FILE IN A RNL COMPATIBLE FORMAT.
```

```
-----------------------------------------------------
*/
strcat(fc,".stim");
fid=fopen(fc,"w");
fprintf(fid, "time_range 0 %d",rr); newline;
fprintf(fid, "carin1 2 1 0 1 1"); newline;
fprintf(fid, "input 0 h 0 h 1 1 2 1 3 1 4"); newline;
fprintf(fid, "c1 2 1 0 h 1"); newline;
fprintf(fid, "report 1 0"); newline;
fclose(fid);
sprintf(memory, "gen_time %s.stim %s.time",fs,fs);
system(memory);
/*
-----------------------------------------------------------------
CREATE A FILE THAT CONTAINS A SEQUENCE OF RNL COMMAND AND SIMULATE.
-----------------------------------------------------------------
*/
strcat(fb,".1");
fid=fopen(fb,"w");
fprintf(fid, "(load \"uwstd.1\")"); newline;
fprintf(fid, "(load \"uwsim.1\")"); newline;
fprintf(fid, "(log-file \"%s.rlog\")",fs); newline;
fprintf(fid, "(read-network \"%s\")",fs); newline;
fprintf(fid, "(setq incr 100)"); newline;
fprintf(fid, "(sim-init)"); newline;
fprintf(fid, "(sim-init)"); newline;
fprintf(fid, "(def-report '(\" \"  ");
for(i=M; i>0; i--){
    fprintf(fid, "ffo%d ",i);
                  }
for(i=N; i>0; i--){
    fprintf(fid, "sum%d ",i);
                  }
fprintf(fid, "))"); newline;
fprintf(fid, "(setq lanalyze t)"); newline;
fprintf(fid, "(wr-format)"); newline;
fprintf(fid, "(load \"%s.time\")",fs); newline;
fprintf(fid, "exit"); newline;
fclose(fid);
sprintf(memory, "rn1 %s.1",fs);
system(memory);
sprintf(memory, "print %s.rlog",fs);
system(memory);
}
```

# APPENDIX B
## Network Description Files

### lib.net - Library File

```
; (1) CMOS INVERTER
; -----------------
; (2) NODES IN THE NETWORK
(node in out)
; (3) P-CHANNEL
(ptrans in out Vdd 8 8)
; (4) N-CHANNEL
(etrans in GND out 4 8)
; (5) CAPACITANCE
(capacitance out 0.03)


; (1)MACRO DEFINITION FOR CMOS LATCH
; ----------------------------------
; (2)NAMING THE MACRO AND ITS PARAMETERS
(macro latch (out in cl cl-)
; (3)DECLARATION OF THE NODES LOCAL TO THE LATCH
(local n1)
; (4)FIRST CLOCKED CMOS INVERTER
(clkinv n1 in cl cl-)
; (5)UNLOCKED CMOS INVERTER
(cinvert out n1)
; (6)SECOND CLOCKED CMOS INVERTER
(clkinv n1 out cl- cl)
; (7)CLOSING PARENTHESIS FOR THE MACRO
)


; (1)MACRO DEFINITION FOR CMOS MASTER SLAVE FLIP-FLOP
; ---------------------------------------------------
; (2)NAMING THE MACRO AND ITS PARAMETER
(macro msff (out in cl)
```

```
; (3)DECLARATION OF THE NODES LOCAL TO FLIP-FLOP


(local n1 n2)

; (4)FIRST LATCH
(latch n1 in c1 n2)

; (5)SECOND LATCH
(latch out n1 n2 c1)

; (6)CMOS INVERTER
(cinvert n2 c1)

; (7)CLOSING PARENTHESIS FOR THE MACRO
)
```

## XOR2.net - 2 Input XOR Gates

```
;macro name
;----------
(macro xor2 (out i1 i2)

;local node
(local i1- i2- p1 p2 p3 p4)

;inverter
;--------
(cinvert i1- i1)
(cinvert i2- i2)

;pull up
;-------
(ptrans i1- p1 Vdd 8 8)
(ptrans i2  out p1 8 8)
(ptrans i1  p2 Vdd 8 8)
(ptrans i2- out p2 8 8)

;pull down
;---------
(etrans i1- p3 out  4 8)
(etrans i2- GND p3 4 8)
(etrans i1  p4 out  4 8)
(etrans i2  GND p4 4 8)

;closing
)
```

## XOR3.net - 3 Input XOR Gates

```
;macro name
;----------
(macro xor3 (out a b c)

;local node
(local a- b- c- p1 p2 p3 p4 p5 p6 p7 p8)

;inverter
;--------
(cinvert a- a)
(cinvert b- b)
(cinvert c- c)

;pull up
;-------
(ptrans a p1 Vdd 8 8)
(ptrans a- p2 Vdd 8 8)
(ptrans a- p3 Vdd 8 8)
(ptrans a p4 Vdd 8 8)
(ptrans b- out p1 8 8)
(ptrans b out p2 8 8)
(ptrans c out p3 8 8)
(ptrans c- out p4 8 8)

;pull down
;---------
(etrans a- p5 out 4 8)
(etrans b- p5 p6 4 8)
(etrans c- GND p6 4 8)
(etrans a p7 out 4 8)
(etrans b p7 p8 4 8)
(etrans c GND p8 4 8)

;closing
)
```

## lfsr8.net - 8 Stage LFSR

```
(load "lib.net")
(load "xor2.net")

(
node cl

 ffout1
 ffout2
 ffout3
 ffout4
 ffout5
 ffout6
 ffout7
 ffout8

 ffin1
 ffin2
 ffin3
 ffin4
 ffin5
 ffin6
 ffin7
 ffin8

  input

 xorout1 xorin11 xorin12
 xorout2 xorin21 xorin22
 xorout3 xorin31 xorin32
 xorout4 xorin41 xorin42
 )

 (msff ffout1 ffin1 cl)
 (msff ffout2 ffin2 cl)
 (msff ffout3 ffin3 cl)
 (msff ffout4 ffin4 cl)
 (msff ffout5 ffin5 cl)
 (msff ffout6 ffin6 cl)
 (msff ffout7 ffin7 cl)
 (msff ffout8 ffin8 cl)

  (xor2 xorout1 xorin11 xorin12 )
  (xor2 xorout2 xorin21 xorin22 )
  (xor2 xorout3 xorin31 xorin32 )
```

```
(xor2 xorout4 xorin41 xorin42 )

(connect ffin8 ffout7)

(connect ffin7 xorout1)
(connect xorin11 ffout6)
(connect xorin12 ffout8)

(connect ffin6 xorout2)
(connect xorin21 ffout5)
(connect xorin22 ffout8)

(connect ffin5 ffout4)

(connect ffin4 xorout3)
(connect xorin31 ffout3)
(connect xorin32 ffout8)

(connect ffin3 ffout2)

(connect ffin2 ffout1)

(connect ffin1 xorout4)
(connect xorin41 ffout8)
(connect xorin42 input)
```

## gadd4.net - 4 Bit Fault-Free Full-Adder

```
(node
a4 b4 carin4 carout4 sum4
a3 b3 carin3 carout3 sum3
a2 b2 carin2 carout2 sum2
a1 b1 carin1 carout1 sum1

invin41 invin42 invin43
invin31 invin32 invin33
invin21 invin22 invin23
invin11 invin12 invin13

invout41 invout42 invout43
invout31 invout32 invout33
invout21 invout22 invout23
invout11 invout12 invout13

norin41 norin42
norin31 norin32
norin21 norin22
norin11 norin12

norout4
norout3
norout2
norout1

xoro41 xoro42
xoro31 xoro32
xoro21 xoro22
xoro11 xoro12

xori411 xori412 xori421 xori422
xori311 xori312 xori321 xori322
xori211 xori212 xori221 xori222
xori111 xori112 xori121 xori122

nandout41 nandout42
nandout31 nandout32
nandout21 nandout22
nandout11 nandout12
```

```
nandin411 nandin412 nandin421 nandin422
nandin311 nandin312 nandin321 nandin322
nandin211 nandin212 nandin221 nandin222
nandin111 nandin112 nandin121 nandin122
)

(xor2 xoro41 xori411 xori412 )
(xor2 xoro42 xori421 xori422 )

(xor2 xoro31 xori311 xori312 )
(xor2 xoro32 xori321 xori322 )

(xor2 xoro21 xori211 xori212 )
(xor2 xoro22 xori221 xori222 )

(xor2 xoro11 xori111 xori112 )
(xor2 xoro12 xori121 xori122 )

(cnand nandout41 nandin411 nandin412 )
(cnand nandout42 nandin421 nandin422 )
(cnand nandout31 nandin311 nandin312 )
(cnand nandout32 nandin321 nandin322 )
(cnand nandout21 nandin211 nandin212 )
(cnand nandout22 nandin221 nandin222 )
(cnand nandout11 nandin111 nandin112 )
(cnand nandout12 nandin121 nandin122 )

(nor norout4 norin41 norin42 )
(nor norout3 norin31 norin32 )
(nor norout2 norin21 norin22 )
(nor norout1 norin11 norin12 )

(cinvert invout41 invin41)
(cinvert invout42 invin42)
(cinvert invout43 invin43)
(cinvert invout31 invin31)
(cinvert invout32 invin32)
(cinvert invout33 invin33)

(cinvert invout21 invin21)
(cinvert invout22 invin22)
(cinvert invout23 invin23)

(cinvert invout11 invin11)
(cinvert invout12 invin12)
```

```
(cinvert invout13 invin13)

(connect a4 xori411)
(connect b4 xori412)
(connect carin4 xori421)
(connect xoro41 xori422)
(connect xoro42 sum4)

(connect a3 xori311)
(connect b3 xori312)
(connect carin3 xori321)
(connect xoro31 xori322)
(connect xoro32 sum3)

(connect a2 xori211)
(connect b2 xori212)
(connect carin2 xori221)
(connect xoro21 xori222)
(connect xoro22 sum2)

(connect a1 xori111)
(connect b1 xori112)
(connect carin1 xori121)
(connect xoro11 xori122)
(connect xoro12 sum1)

(connect a4 nandin411)
(connect b4 nandin412)
(connect nandout41 invin41)

(connect a3 nandin311)
(connect b3 nandin312)
(connect nandout31 invin31)

(connect a2 nandin211)
(connect b2 nandin212)
(connect nandout21 invin21)

(connect a1 nandin111)
(connect b1 nandin112)
(connect nandout11 invin11)

(connect xoro41 nandin421)
(connect carin4 nandin422)
```

```
(connect nandout42 invin42)
(connect xoro31 nandin321)
(connect carin3 nandin322)

(connect nandout32 invin32)
(connect xoro21 nandin221)
(connect carin2 nandin222)

(connect nandout22 invin22)
(connect xoro11 nandin121)
(connect carin1 nandin122)

(connect nandout12 invin12)

(connect invout41 norin41)
(connect invout42 norin42)
(connect invout31 norin31)
(connect invout32 norin32)
(connect invout21 norin21)
(connect invout22 norin22)
(connect invout11 norin11)
(connect invout12 norin12)

(connect norout4 invin43)
(connect invout43 carout4)

(connect norout3 invin33)
(connect invout33 carout3)

(connect norout2 invin23)
(connect invout23 carout2)

(connect norout1 invin13)
(connect invout13 carout1)

(connect carin4 carout3)
(connect carin3 carout2)
(connect carin2 carout1)
```

## badd40.net - 4 Bit s-a-0 Fault Full-Adder

```
(load "lib.net")
(load "xor2.net")

(node
a4 b4 carin4 carout4 sum4
a3 b3 carin3 carout3 sum3
a2 b2 carin2 carout2 sum2
a1 b1 carin1 carout1 sum1

invin41 invin42 invin43
invin31 invin32 invin33
invin21 invin22 invin23
invin11 invin12 invin13

invout41 invout42 invout43
invout31 invout32 invout33
invout21 invout22 invout23
invout11 invout12 invout13

norin41 norin42
norin31 norin32
norin21 norin22
norin11 norin12

norout4
norout3
norout2
norout1

xoro41 xoro42
xoro31 xoro32
xoro21 xoro22
xoro11 xoro12

xori411 xori412 xori421 xori422
xori311 xori312 xori321 xori322
xori211 xori212 xori221 xori222
xori111 xori112 xori121 xori122

nandout41 nandout42
nandout31 nandout32
nandout21 nandout22
nandout11 nandout12
```

```
nandin411 nandin412 nandin421 nandin422
nandin311 nandin312 nandin321 nandin322
nandin211 nandin212 nandin221 nandin222
nandin111 nandin112 nandin121 nandin122
)

(xor2 xoro41 xori411 xori412 )
(xor2 xoro42 xori421 xori422 )

(xor2 xoro31 xori311 xori312 )
(xor2 xoro32 xori321 xori322 )

(xor2 xoro21 xori211 xori212 )
(xor2 xoro22 xori221 xori222 )

(xor2 xoro11 xori111 xori112 )
(xor2 xoro12 xori121 xori122 )

(cnand nandout41 nandin411 nandin412 )
(cnand nandout42 nandin421 nandin422 )
(cnand nandout31 nandin311 nandin312 )
(cnand nandout32 nandin321 nandin322 )
(cnand nandout21 nandin211 nandin212 )
(cnand nandout22 nandin221 nandin222 )
(cnand nandout11 nandin111 nandin112 )
(cnand nandout12 nandin121 nandin122 )

(nor norout4 norin41 norin42 )
(nor norout3 norin31 norin32 )
(nor norout2 norin21 norin22 )
(nor norout1 norin11 norin12 )

(cinvert invout41 invin41)
(cinvert invout42 invin42)
(cinvert invout43 invin43)

(cinvert invout31 invin31)
(cinvert invout32 invin32)
(cinvert invout33 invin33)

(cinvert invout21 invin21)
(cinvert invout22 invin22)
(cinvert invout23 invin23)

(cinvert invout11 invin11)
```

```
(cinvert invout12 invin12)
(cinvert invout13 invin13)

(connect a4 xori411)
(connect b4 xori412)
(connect carin4 xori421)
(connect xoro41 xori422)
(connect xoro42 sum4)

(connect a3 xori311)
(connect b3 xori312)
(connect carin3 xori321)
(connect xoro31 xori322)
(connect xoro32 sum3)

(connect a2 xori211)
(connect b2 xori212)
(connect carin2 xori221)
(connect xoro21 xori222)
(connect xoro22 sum2)

(connect a1 xori111)
(connect b1 xori112)
(connect GND xori121)
(connect xoro11 xori122)
(connect xoro12 sum1)

(connect a4 nandin411)
(connect b4 nandin412)
(connect nandout41 invin41)

(connect a3 nandin311)
(connect b3 nandin312)
(connect nandout31 invin31)

(connect a2 nandin211)
(connect b2 nandin212)
(connect nandout21 invin21)

(connect a1 nandin111)
(connect b1 nandin112)
(connect nandout11 invin11)

(connect xoro41 nandin421)
(connect carin4 nandin422)
```

```
(connect nandout42 invin42)
(connect xoro31 nandin321)
(connect carin3 nandin322)

(connect nandout32 invin32)
(connect xoro21 nandin221)
(connect carin2 nandin222)

(connect nandout22 invin22)
(connect xoro11 nandin121)
(connect carin1 nandin122)

(connect nandout12 invin12)

(connect invout41 norin41)
(connect invout42 norin42)
(connect invout31 norin31)
(connect invout32 norin32)
(connect invout21 norin21)
(connect invout22 norin22)
(connect invout11 norin11)
(connect invout12 norin12)

(connect norout4 invin43)
(connect invout43 carout4)

(connect norout3 invin33)
(connect invout33 carout3)

(connect norout2 invin23)
(connect invout23 carout2)

(connect norout1 invin13)
(connect invout13 carout1)

(connect carin4 carout3)
(connect carin3 carout2)
(connect carin2 carout1)
```

## badd41.net - 4 Bit s-a-1 Fault Full-Adder

```
(load "lib.net")
(load "xor2.net")

(node
a4 b4 carin4 carout4 sum4
a3 b3 carin3 carout3 sum3
a2 b2 carin2 carout2 sum2
a1 b1 carin1 carout1 sum1

invin41 invin42 invin43
invin31 invin32 invin33
invin21 invin22 invin23
invin11 invin12 invin13

invout41 invout42 invout43
invout31 invout32 invout33
invout21 invout22 invout23
invout11 invout12 invout13

norin41 norin42
norin31 norin32
norin21 norin22
norin11 norin12

norout4
norout3
norout2
norout1

xoro41 xoro42
xoro31 xoro32
xoro21 xoro22
xoro11 xoro12

xori411 xori412 xori421 xori422
xori311 xori312 xori321 xori322
xori211 xori212 xori221 xori222
xori111 xori112 xori121 xori122

nandout41 nandout42
nandout31 nandout32
nandout21 nandout22
nandout11 nandout12
```

```
nandin411 nandin412 nandin421 nandin422
nandin311 nandin312 nandin321 nandin322
nandin211 nandin212 nandin221 nandin222
nandin111 nandin112 nandin121 nandin122
)

(xor2 xoro41 xori411 xori412 )
(xor2 xoro42 xori421 xori422 )

(xor2 xoro31 xori311 xori312 )
(xor2 xoro32 xori321 xori322 )

(xor2 xoro21 xori211 xori212 )
(xor2 xoro22 xori221 xori222 )

(xor2 xoro11 xori111 xori112 )
(xor2 xoro12 xori121 xori122 )

(cnand nandout41 nandin411 nandin412 )
(cnand nandout42 nandin421 nandin422 )
(cnand nandout31 nandin311 nandin312 )
(cnand nandout32 nandin321 nandin322 )
(cnand nandout21 nandin211 nandin212 )
(cnand nandout22 nandin221 nandin222 )
(cnand nandout11 nandin111 nandin112 )
(cnand nandout12 nandin121 nandin122 )

(nor norout4 norin41 norin42 )
(nor norout3 norin31 norin32 )
(nor norout2 norin21 norin22 )
(nor norout1 norin11 norin12 )

(cinvert invout41 invin41)
(cinvert invout42 invin42)
(cinvert invout43 invin43)

(cinvert invout31 invin31)
(cinvert invout32 invin32)
(cinvert invout33 invin33)

(cinvert invout21 invin21)
(cinvert invout22 invin22)
(cinvert invout23 invin23)

(cinvert invout11 invin11)
```

```
(cinvert invout12 invin12)
(cinvert invout13 invin13)

(connect a4 xori411)
(connect b4 xori412)
(connect carin4 xori421)
(connect xoro41 xori422)
(connect xoro42 sum4)

(connect a3 xori311)
(connect b3 xori312)
(connect carin3 xori321)
(connect xoro31 xori322)
(connect xoro32 sum3)

(connect a2 xori211)
(connect b2 xori212)
(connect carin2 xori221)
(connect xoro21 xori222)
(connect xoro22 sum2)

(connect a1 xori111)
(connect b1 xori112)
(connect Vdd xori121)
(connect xoro11 xori122)
(connect xoro12 sum1)

(connect a4 nandin411)
(connect b4 nandin412)
(connect nandout41 invin41)

(connect a3 nandin311)
(connect b3 nandin312)
(connect nandout31 invin31)

(connect a2 nandin211)
(connect b2 nandin212)
(connect nandout21 invin21)

(connect a1 nandin111)
(connect b1 nandin112)
(connect nandout11 invin11)

(connect xoro41 nandin421)
(connect Vdd nandin422)
```

```
(connect nandout42 invin42)
(connect xoro31 nandin321)
(connect carin3 nandin322)

(connect nandout32 invin32)
(connect xoro21 nandin221)
(connect carin2 nandin222)

(connect nandout22 invin22)
(connect xoro11 nandin121)
(connect carin1 nandin122)

(connect nandout12 invin12)

(connect invout41 norin41)
(connect invout42 norin42)
(connect invout31 norin31)
(connect invout32 norin32)
(connect invout21 norin21)
(connect invout22 norin22)
(connect invout11 norin11)
(connect invout12 norin12)

(connect norout4 invin43)
(connect invout43 carout4)

(connect norout3 invin33)
(connect invout33 carout3)

(connect norout2 invin23)
(connect invout23 carout2)

(connect norout1 invin13)
(connect invout13 carout1)

(connect carin4 carout3)
(connect carin3 carout2)
(connect carin2 carout1)
```

## misr5.net - 5 Stage MISR

```
(load "lib.net")
(load "xor2.net")
(load "xor3.net")

(
node r cl

 ffo1
 ffo2
 ffo3
 ffo4
 ffo5

 ffi1
 ffi2
 ffi3
 ffi4
 ffi5

 in1
 in2
 in3
 in4
 in5

xorout21 xorin211 xorin212
xorout22 xorin221 xorin222
xorout23 xorin231 xorin232
xorout24 xorin241 xorin242

xorout31 xorin311 xorin312 xorin313
)

(msff ffo1 ffi1 cl)
(msff ffo2 ffi2 cl)
(msff ffo3 ffi3 cl)
(msff ffo4 ffi4 cl)
(msff ffo5 ffi5 cl)

(xor2 xorout21 xorin211 xorin212 )
(xor2 xorout22 xorin221 xorin222 )
(xor2 xorout23 xorin231 xorin232 )
(xor2 xorout24 xorin241 xorin242 )
```

```
(xor3 xorout31 xorin311 xorin312 xorin313)

(connect ffi5 xorout21)
(connect xorin211 ffo4)
(connect xorin212 in5)

(connect ffi4 xorout22)
(connect xorin221 ffo3)
(connect xorin222 in4)

(connect ffi3 xorout31)
(connect xorin311 ffo2)
(connect xorin312 ffo5)
(connect xorin313 in3)

(connect ffi2 xorout23)
(connect xorin231 ffo1)
(connect xorin232 in2)

(connect ffi1 xorout24)
(connect xorin241 ffo5)
(connect xorin242 in1)
```

# LIST OF REFERENCES


1. Williams, T. W., *VLSI Testing*, Elsevier Science Publishers, B.V. (North-Holland), pp. 29-30, May 1986.

2. Bhaskar, K. S., *Signature Analysis: Yet Another Perspective*, International Test Conference, pp. 132-134, November 1982.

3. Zuxi, Sun, "Self-Testing of Embedded RAMs", International Test Conference, October 1984, pp. 148-156.

4. Damiani, M., Olivio, P., and Favalli, M., "Aliasing in Signature Analysis Testing With Multiple-Input Shift Register", European Test Conference, April 1989, pp. 346-353.

5. Abadir, M. S., and Regubati, H. K., "LSI Testing Techniques", IEEE Micro, February 1983, pp. 34-51.

6. Breuer, M. A., and Carlan, A. J., "State of the Art Assessment of Testing and Testability of LSI/VLSI Circuits", Engineering Group Report, October 1982, pp. 10-11.

7. Karpovsky, M., "Universal Tests Detecting Input/Output Faults in Almost All Devices", International Test Conference, November 1982, pp. 52-53.

8. William, T. H., and Parker, K. P., "Design for Testability - A Survey", *Proceedings of the IEEE*, Vol. 71, No. 1, January 1983, pp. 98-112.

9. El-Zig, Y. M., and Cloutier, R. J., "Functional Level Test Generation for Stuck-Open Faults in CMOS VLSI", International Test Conference, October 1981, pp. 536-546.

10. Roth, J. P., "Diagnosis of Automata Failures: A Calculus and A Method", *Journal of Research and Development*, Vol. 10, October 1966, pp. 278-291.

11. Bardell, P. H., *Built-in Test for VLSI*, John Wiley & Sons, August 1987, pp. 10-11.

12. Weste, N., and Eshraghian, K., *Principles of CMOS VLSI Design, A System Perspective*, Addison-Wesley, Reading, Massachusetts, 1985.

13. McCluskey, E. J., and Wang, L. T., "Built-in Self Test for Sequential Machines", International Test Conference, September 1987, pp. 334-340.

14. Koenemann, B., Mucha, J., and Zwiehoff, G., "Built-in Logic Block Observation Techniques", International Test Conference, October 1979, pp. 37-41.

15. "Signature Analysis", *Hewlett-Packard Journal*, Vol. 28, No. 9, May 1977.

16. Daehn, W., Williams, T. W., and Wagner, K. D., "Aliasing Errors in Linear Automata Used as Multiple Input Signature Analyzer", *IBM Journal Research Development*, Vol. 34, Nos. 2/3, March/May 1990, pp. 363-364.

17. Hiawiczka, A., "Hybrid Design of Parallel Signature Analyzers", European Test Conference, April 1989, pp. 354-355.

18. Hassan, S. Z., and McCluskey, E. J., "Pseudo-Exhaustive Testing of Sequential Machine Using Signature Analysis", International Test Conference, 1984, pp. 320-326.

19. William, T. W., and Daehn, W., "Aliasing Errors with Primitive and Non-Primitive Polynomials", International Test Conference, September 1987, pp. 639-640.

20. Hortensius, P. D., McLeod, R. D., and Podaima, B. W., "Cellular Automata Circuits for Built-in Self Test", *IBM Journal Research Development*, Vol. 34, No. 2/3, March/May 1990, pp. 389-392.

21. Tsui, Frank F., *LSI/VLSI Testability*, McGraw-Hill, New York, 1987, pp. 173-175.

22. Tanenbaum, A. X., *Structured Computer Organization*, Prentice-Hall, Inc., 1990, pp. 96-98.

23. Chin, Miao, "A CMOS CRC Divider Generator", Master's Thesis, Naval Postgraduate School, 1991, pp. 10-12.

24. "VLSI Design Tools Reference Manual, Release 3.2", NW Laboratory for Integrated Systems, September 1988, pp. 5-3.

# INITIAL DISTRIBUTION LIST

| | | No. of Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145 | 2 |
| 2. | Library, Code 52<br>Naval Postgraduate School<br>Monterey, CA 93943-5002 | 2 |
| 3. | Chairman, Code EC<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 1 |
| 4. | Professor Chyan Yang, Code EC/Ya<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 2 |
| 5. | Professor Herschel H. Loomis, Jr., Code EC/Lm<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 1 |
| 6. | Dirdikau<br>MABESAU<br>Jl: Gatot Subroto 72<br>Jakarta, Selatan<br>Indonesia | 1 |
| 7. | Dirlekau<br>MABESAU<br>Jl: Gatot Subroto 72<br>Jakarta, Selatan<br>Indonesia | 1 |

8. Kadispullantau
MABESAU
Jl: Gatot Subroto 72
Jakarta, Selatan
Indonesia — 1

9. Col. Nav Marajuki
Komplex CURAG INDAH
Jl: ELANG MALINDO iX/C3 No:22
Jakarta, Timur
Indonesia — 1

10. Jurusan Matematika
FIPPA - UNPAD
Jl: Raya Bandung - Sumedang KM21
Jatinangor
Sumedang
Jawa Barat
Indonesia — 1

11. Jasa Barus
Dispullahta
Mabesau
Jl: Gatot Subroto 72
Jakarta, Selatan
Indonesia — 2